Titre: Title:	Dynamic Enforcement of Security Policies in Multi-Tenant Cloud Networks
Auteur: Author:	Tommy Koorevaar
Date:	2012
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Koorevaar, T. (2012). Dynamic Enforcement of Security Policies in Multi-Tenant Cloud Networks [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1056/

UNIVERSITÉ DE MONTRÉAL

DYNAMIC ENFORCEMENT OF SECURITY POLICIES IN MULTI-TENANT CLOUD NETWORKS

TOMMY KOOREVAAR
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
NOVEMBRE 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DYNAMIC ENFORCEMENT OF SECURITY POLICIES IN MULTI-TENANT CLOUD NETWORKS

présenté par : KOOREVAAR Tommy
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de :

M. FERNANDEZ José M., Ph.D., président
M. PIERRE Samuel Ph.D., membre et directeur de recherche
M. QUINTERO Alejandro Doct., membre

à ma famille,
à mes amis.

ACKNOWLEDGEMENTS

I would like to thank first my advisor, Samuel Pierre, for believing in me and guiding me throughout my master's. Thanks to him, I worked on an inspiring research project, close to passionate researchers.

I would also like to thank Ericsson Research Canada, and particularly Makan Pourzandi, for his precious advice both on a professional and personal level. His guidance was crucial in the realization of my research project.

Finally, I would like to thank all members of the LARIM. Its friendly and relaxed atmosphere as well as the kindness and helpfulness of its members has put me in the best conditions possible.

RÉSUMÉ

Au cours des dernières années, l'évolution des nouvelles technologies a changé notre façon de travailler. Les grandes entreprises, les gouvernements et même nous en tant qu'individus dépendons des ordinateurs et des réseaux. Ils sont devenus une partie importante de nos vies personnelle et professionnelle, et représentent maintenant une infrastructure critique au même titre que les réseaux électriques, tant la quantité de données numériques et devenue importante.

Cette évolution continue avec la montée en puissance de l'informatique en nuage. Dans ce nouveau modèle, on peut accéder à distance à des logiciels, à du stockage numérique ou bien à des infrastructures sans contrainte, les machines étant regroupées en centre de données dont l'accès se fait de manière transparente par internet.

La sécurité de l'information est devenue primordiale en informatique, et en particulier dans l'informatique en nuage au fur et à mesure que les entreprises y exportent leurs données sensibles. Ainsi, le niveau de sécurité au sein des centres de données doivent être au moins équivalent à ce que les entreprises ont dans leur propres installations.

Nous appellerons middlebox, un élément du réseau ayant pour fonction d'inspecter et de filtrer les paquets, dans un but autre que la retransmission de paquet. Un pare-feu est un bon exemple de middlebox.

Les solutions existantes pour sécuriser l'informatique en nuage prennent rarement en considération la traversée de middleboxes, en effet, elles se concentrent principalement sur l'isolation des trafics entre les différents clients du centre de données. De plus, les solutions prenant en compte l'application de middlebox le font d'une façon qui ne permet pas la migration des nœuds au sein du réseau.

Notre projet consiste en la création d'une architecture permettant l'application de politiques de sécurité par client. Les politiques de sécurité seront des séquences de middlebox que le trafic des clients devra traverser, puisque c'est de cette façon que la sécurité est généralement assurée dans les entreprises. L'application de ces politiques de sécurité devra prendre en compte la pluralité des clients ainsi que la migration des machines au sein du réseau. Plus précisément, le trafic devra traverser les middlebox dans l'ordre spécifié par le client, sans en traverser d'autres. L'application des politiques doit être automatiquement re-

configurée lors de la migration des machines virtuelles.

Afin de réaliser ce projet, nous utilisons une architecture de réseau programmable afin d'appliquer efficacement les politiques. Dans cette architecture, le plan de contrôle est découplé du plan de données ce qui permet de centraliser la gestion du réseau. Ainsi, les clients de notre architecture définissent le niveau de sécurité qu'ils veulent voir être appliqué à leur machines. Afin d'identifier une politique de sécurité, nous utilisons un identifiant d'application (*AppID*), qui représente une chaîne de middlebox à traverser.

Nous supposons que l'hyperviseur a la capacité d'insérer cet *AppID* au sein des paquets, lorsqu'une machine en émet. Au moment où le premier paquet d'un flux atteint un commutateur du réseau, il est transférer au contrôleur de réseau, qui a pour tâche de lire cet *AppID*. En fonction de ce dernier, le contrôleur détermine la chaîne de sécurité à appliquer au trafic.

Nous définissons différents marqueurs (que nous nommons *EEL-tags*) afin d'effectuer le routage des paquets au travers des middlebox. Ceux-ci sont subdivisés en gTag et iTag. Les gTags correspondent à des types de middlebox, tandis que les iTags correspondent à des instances de middlebox. Ainsi, une chaîne de sécurité est définie par une chaîne de gTag. Les itags correspondants sont utilisés afin d'effectuer le routage des paquets au sein du réseau, en définissant la prochaine instance à traverser.

En ajoutant ces *EEL-tags* aux paquets, notre modèle procure une façon simple et automatique d'appliquer des politiques de sécurité, tout en s'assurant de leur cohérence malgré la migration des nœuds. De plus, notre modèle permet au réseau d'être divisé en petites zones, chacune d'entre elle étant contrôlée par un contrôleur de réseau spécifique. Lorsque les machines émettrice et réceptrice se trouvent dans deux zones différentes, l'application de la politique de sécurité peut être répartie entre les différentes zones.

Nous avons développé un prototype que nous avons testé dans un environnement simulé. Bien que de nombreux aspects de notre implémentation requièrent de l'amélioration afin d'obtenir une solution commerciale, cette expérimentation nous a permis d'obtenir une preuve de concept de notre architecture. Nous avons notamment pu observer que les politiques de sécurité restent cohérentes malgré la migration de nœuds.

ABSTRACT

During the past decades, the evolution of technology has drastically changed our ways. All major enterprises, government services, and even us as individuals, rely on computers and networks. They have become a part of our personal and professional lives and represent nowadays a critical infrastructure as the amount of data stored numerically as well as its sensitivity has grown considerably.

This evolution continues with the rise of cloud computing. In this new model, one can access software, digital storage or infrastructure without constraints, as the hardware is pooled in remote data center, accessed seamlessly via Internet.

Security has become a major concern in computer science in general and in the cloud in particular, as enterprises moving to the cloud would have to export some of their sensitive data. Therefore, the cloud providers need to offer a level of security which matches what the companies have in their on-site installations.

A middlebox is a network appliance that inspects and filters packets for purposes other than packet forwarding. A firewall is a good example of a middlebox.

Existing solutions to secure the cloud rarely take in consideration the traversal of middleboxes, as they focus mainly on creating an isolation between the different tenants. Furthermore, the solutions considering the traversal of middlebox sequences do so in a way which does not permit the migration of nodes.

Through our project, we aim to create a cloud architecture allowing the application of security policies per tenant. The security will consist in sequences of middleboxes to be traversed, as it is the way commonly used by enterprises to secure their networks. The enforcement of security policies will have to take in consideration the multi-tenant aspect of the cloud, as well as the node migration.

Particularly, traffic should traverse middleboxes in the sequence required by the tenant and should not traverse unnecessary middleboxes. The enforcement of policies should be automatically re-configured due to VM migrations.

In this work, we propose a method of leveraging the current Software-Defined Network (SDN) architecture for efficient policy enforcement. SDN is a form of network architecture

in which the control plane is separated from the data plane, allowing the network to be centrally managed. Therefore, the tenants define the security design they want to apply to their Virtual Machine (VM)s, or groups of VMs. In order to identify the security policies, we use an Application ID (AppID), which actually refers to a chain of middleboxes to be traversed.

We assume that the running hypervisor has the capability to add this *AppID* into the flow when a VM emits packets. When the first packet of a flow reaches a switch, it is forwarded to the network controller, which in turn retrieves the *AppID* from the packet. Based on the *AppID*, the controller determines the chain of middleboxes to be traversed.

In order to route the packets through the middleboxes, our model defines labels to apply to each flow of packets (*EEL-tags*). The latter are divided in generic EEL-tag (gTag) and instance EEL-tag (iTag). Each gTag corresponds to a middlebox type, and each iTag corresponds to a middlebox instance. The security chain is defined by a chain of gTags. The iTags are added to the packets in order to route the packets across the network, defining what is the next middlebox the packet must be sent to.

By using the *EEL-tags*, this model provides a simple way to automatically enforce security policies, while keeping them consistent despite node migration. Furthermore, we allow the network to be partitioned in different zones, each zone being ruled by a specific controller. When the VM source and destination belong to different zones, the enforcement of security policies can be spread between the different zones.

We created a prototype of our model that we tested in a simulated environment. Although many aspects of our implementation will have to be improved in order to obtain a viable commercial solution, testing our prototype provided us with a proof of concept. Particularly, it showed how the security policies remain consistent despite node migration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ANNEXES

LIST OF ABBREVIATIONS

EEL	Elastic Enforcement Layer
iTag	instance EEL-tag
gTag	generic EEL-tag
OF	OpenFlow
OFC	OpenFlow-Controller
OFS	OpenFlow-Switch
VM	Virtual Machine
IDS	Intrusion Detection System
DPI	Deep Packet Inspection
AppFW	Application Firewall
AppID	Application ID
SDN	Software-Defined Network
VLAN	Virtual Local Area Network
LAN	Local Area Network
gTags	Generic Tags
iTags	Instance Tags
IEEE	Institute of Electrical and Electronics Engineers
IaaS	Infrastructure as a service
PaaS	Platform as a service
SaaS	Software as a service
B2B	Business to Business
B2C	Business to Customer
dpid	Datapath-ID

CHAPTER 1

INTRODUCTION

1.1 Definitions and Concepts

Security has become a major concern during the past decade, as computers and networks are becoming a critical infrastructure. All major enterprises and government services, and even us as individuals, rely on them. The amount of data stored digitally as well as its sensitivity has likewise grown considerably.

Thereby, security breaches lead to disastrous business and legal consequences, now more than ever. In this very first chapter, we are defining several terms which are important to our project.

1.1.1 Middlebox

A middlebox is a network appliance performing functions other than standard functions of an IP router or a switch. It lies between the source and the destination and transforms, inspects, filters, or manipulates traffic in order to provide services such as Network Address translation, Intrusion Detection or Firewall services.

1.1.2 Cloud Computing

The idea of cloud computing is almost as old as the computer itself. Its principle is to have the user's computer, Smartphone, tablet or any internet-connected device acting as a front-end displaying an application, as the resources used to do so are in fact located in remote servers.

These remote servers are often Virtual Machine (VM) located on physical servers inside data centers. Companies specialized in leasing their data centers are called cloud providers. This leasing can be done both in a Business to Business (B2B) or Business to Customer (B2C) model.

The cloud providers are often classified in three main categories, depending on the model of service they are offering [28]. The first model is the Infrastructure as a service (IaaS), which is the most basic model. The IaaS model consists in providing Virtual Machines to the clients whom will have to install operating systems as well as their applications on top of the VMs. Amazon EC2, Google Compute Engine and Rackspace Cloud are examples

of IaaS. Secondly, there is the Platform as a service (PaaS) model where only a computing platform is provided to the client, on top of which the client's application is directly installed. Examples of PaaS include Microsoft Azure and Google App Engine. Finally, the last model is the Software as a service (SaaS), which is the most widely known among internet users. In this model, the end-user is using directly the software installed by the cloud provider, which could be an online e-mail service, for instance. This last model differs slightly to the previous as the cloud users are the customers of the application, in comparison of the two first models where the cloud users are often businesses. The Google Apps are a good example of the last model.

In the IaaS model, the customers to dynamically scale up and down to as many machines as needed inside the cloud in a "pay-as-you-go" manner. The clients can dynamically provision resources to meet the current demand by adjusting their leasing of resources from the cloud provider. Customers can also then use more efficient hardware without being preoccupied by its maintenance, cooling and storage.

In each model, the cloud providers can use multi-tenancy, where virtual machines from multiple customers can share the same sets of physical servers and the network, in order to reduce the waste of resources. However, economies of scale are so important that their users benefit from both a more efficient and cheaper solution.

Due to the combination of an ever growing number of people connected to the internet, the constant evolution of computer science along with the decreasing of hardware prices, the use of cloud computing instead of classic networks architecture has recently become a reality.

1.1.3 Software-Defined Networking

Cloud computing is not the only change that has occurred during the past few years, the way we approach networking has also begun to shift. Indeed, as a recent network architecture proposal has drawn significant interests from both academic and industry [11]. In this new architecture, the control plane is decoupled from the forwarding plane and the entire routing system is built as a distributed system. The control plane, running on one or more servers in the network, is overseeing a set of simple forwarding elements.

Traditional routers follow an integrated design where the control plane and data forwarding engine are tightly coupled in the same box. It usually results in overly complicated control plane and complex network management. Due to this high complexity, equipment vendors and network operators are reluctant to employ changes and the network itself is fragile and hard to manage. This is known to create a large burden and high bearer to the new protocol and technology developments.

In this new architecture, the logic coded in the controllers define the way they instruct

the switches forwarding behavior by adding and removing flow entries to switches flow tables. Each flow entry contains a set of actions such as modifying, forwarding or dropping packets. The main task of a forwarding element is then to apply these actions to the matching packets. Therefore, the forwarding logic is completely centralized, which makes it easier to program the behavior of the whole network as a single entity. This is why one often refers to this new architecture as Software-Defined Network or Software-Defined Network (SDN).

1.2 Security Problems in the Cloud

With all new technologies comes a load of uncertainties. Despite the promising potentials of the cloud model, one of the major burden to its widespread adoption is security, as customers are often reluctant to export sensitive data or use computation power that reside outside of their network's frontiers as it would make them accessible to persons foreign to the company. It is common to believe that things are safer in our possession, even though enterprise networks are often less secure than the cloud provider ones [26] as they can often dedicate more resource to the security.

1.2.1 Security Threats

More precisely, cloud computing security can be breached by several actors. In the cloud, there are the cloud provider and the tenants. A tenant could be attacked by the cloud provider or by other clients. The cloud provider could be attacked by tenants. Our project does not consider the situation where the provider is malicious. Indeed, we suppose that the cloud provider is honest. We further suppose that all network appliances are compliant and secure.

Our project aims to secure the traffic of a tenant. In this situation, this traffic could first be threatened by other tenants. Indeed, malicious tenants could be willing to access data of other tenants or to gain access to their network by using techniques such as ARP cache poisoning or IP spoofing. Furthermore, a misconfiguration of routing appliances could lead to a breach in confidentiality. The other threat comes from inside the tenant's network itself. This threat is the most important one in security today, as malicious software can enter the enterprise network downloaded by the employees browsing the internet.

In the enterprise network, the traffic is secured by the traversal of middleboxes along the path. Our project aims to create a framework enabling the traversal of middleboxes despite the elastic nature of the cloud, meanwhile providing isolation of traffic in order to prevent the risk of an attack by another tenant.

1.2.2 Security Requirements

The threat analysis leads us to define requirements for an efficient security solution.

Isolation. The multi-tenant nature of the cloud raises a lot of concerns, since a direct competitor to the company could be using resources of the very same physical server. The cloud provider has thus to ensure a complete isolation between the different tenants. This isolation have to be guaranteed firstly at the hypervisor level, as several VMs should be running on the same physical server without ever accessing each other's data, or affecting the performance of one another. Also, the isolation must be ensured at the network level as we must prevent rogue VMs to be able to sniff packets intended to other VMs. In our work, the isolation at the hypervisor will be assumed but considered out of scope.

Security in the Enterprise Network. Network security is a key aspect in designing modern applications. The policy management in a secure enterprise network today can therefore be quite complex. For instance, it may require restricting a machine containing sensitive data to be accessed only by a small group of users, or preventing external traffic from directly reaching internal servers. The actual realization may involve servers having complex communication patterns governed by network access control, such as the traversal of several middleboxes before being reached. When enterprises decide to move to the cloud, they want to keep the same requirements regarding their policy management. It would be possible for the network manager of the enterprise to implement the middleboxes and the routing policies on VMs in the same way as before moving to the cloud. However, one of the goal of moving to the cloud is to escape the burden of network administration and configuration. Furthermore, the type of security policies in place in enterprises networks are often quite similar as it consists in the traversal of several middleboxes.

Scalability. Today's data centers are already containing hundreds of thousands of servers [7], and this number is very likely to increase during the forthcoming years. Furthermore, the customers can request the creation or the removal of VMs in order to meet their needs. Therefore, the networks are highly elastic and can reach an immense number of VMs. Because of that, networks architecture has to be scalable in order to ensure a good performance whatever number of VM is running in the network.

Node Migration. In order to optimize the resources in the data centers or for maintenance reasons, cloud providers often have to migrate VMs from a physical server to another. Migrations are used in order to pool the active VMs and then reduce the number of turned on

physical servers inside the data center, thus reducing cost. Migrations also reveal themselves handy when there is an hardware problem on a physical server, for example. The VMs running on this server are moved to other physical servers and the alleged deficient server can be turned off and replaced. The impact of node migration on the network is important. Indeed, since the VMs are potentially moving from a physical server to another, the routes between them are also impacted, and the topology of the network is thus changing. Most importantly, the end points of the routes are changing. This causes the need to reconfigure the network with each migration. The cloud provider must ensure that the routes are consistent with the roles and permissions of each VM.

Elastic Networks. Due to recent changes in the use of computers and Internet, the network architecture has drastically changed. It appears now that in data center, the networks have become huge elastic networks which can scale up and down as the tenants provision VMs to meet their needs. Furthermore, the VMs themselves migrate from one physical servers to another, which makes the network volatile in addition to be elastic. We are now dealing with networks that are different from classic ones, as the nodes do not have as many constraints regarding the number of nodes as well as their positioning. Manual configuration of such networks would be time consuming and not responsive enough. It could furthermore lead to misconfiguration and/or malicious actions. An automated way of managing the resources is thus essential.

1.3 Purpose of the Research

It is in this context that our work takes place. We have seen that security is a major hurdle to the cloud adoption, and that the evolution of the network configuration and characteristics makes it hard to configure and administrate.

When tenants decide to migrate to the cloud, they want to keep a similar security level, and it can become complex to implement when their policies are transposed in the cloud. Due to the versatile nature of these elastic networks, it is not possible to implement these routes manually in a similar way to what is done in the enterprises, as the number of routes would be way too important. The manpower needed would be too important and it may lead to misconfiguration and errors. Furthermore, it would not be reactive enough to cope with the constant provision of new VM as well as the live migrations.

As a result, the goal of our research is to :

1. Design a network architecture allowing the automatic enforcement of security policies in a cloud environment ;

2. Implement a prototype of our proposed solution ;
3. Study the viability of our solution by testing the behavior of our prototype.

1.4 Outline

In this introduction, we exposed the context in which this research is made, as well as the problem raised by a massive cloud adoption towards which we may be heading. In the next chapter, we will provide a state of the art of the different ways currently possible to distinguish distinct tenants traffic and apply security policies at the network level. Building on this analysis of the current technologies, we will present the algorithms and models that we produced in order to meet the requirements inherent to our project. After the theory comes the practice, as we will explain our implementation choices for the prototype as well as for the test strategy. This chapter will also hold the proof of concept we obtained. Lastly, the ultimate chapter will be a retrospective on the work that has been done and will expose the conclusions we draw on our project.

CHAPTER 2

TRAFFIC SEPARATION AND SECURITY POLICIES

The current chapter provides a literature review of the main techniques that can be used in the cloud in order to provide a security solution for the cloud's tenants. As evoked in the previous chapter, cloud computing has rapidly risen and is now an important trend as many believe it is the future of computer use. However, this raises many concerns regarding the security around the cloud. As a consequence, the research and development has been intense and the technology has been dramatically evolving.

Across our analysis, we will provide a detailed description of the technologies, by presenting both their practical and technical aspects. We will expose their drawbacks and advantages relatively to the point we are discussing, before giving our conclusions on their performance in a cloud context, and what would be their impact on both the cloud provider and customer.

In a cloud environment, we have seen that any architecture must provide enough scalability in order to operate as the network is scaling up and down. Furthermore, the process has to be automatic in order to reduce the error-prone human interventions. Last, but not least, comes the need of an architecture being migration tolerant as the node migration is a particular characteristic of the cloud.

Our goal is to provide the cloud's tenants with a security solution. In order to do so, we want to distinguish and isolate the tenants' traffic from one another, meanwhile enabling the application of security policies in the form of middlebox sequences to traverse ; this is why the literature review is articulated as follow.

In the first part, we will analyze the different ways existing in order to distinguish the traffics in a multi-tenant infrastructure. We will first review the solutions achieving the isolation based on network separation and then, the ones which are using the traffic's characteristics to distinguish the different flows.

In the second part, we wonder what's the best way to apply security policies. In order to do so, we first analyze the different ways to define a security policy. Subsequently, we will review the work on the enforcement of these policies.

Finally, the conclusion will summarize and close the chapter, recapitulating the main drawbacks in actual solutions while enlightening the key improvement that need to be achieved in order to secure the cloud at the network level while providing distinct security policies to each tenant.

2.1 Traffic Separation

Our goal is to provide security policies to the tenants of the data center. Before defining what a security policy is and how to enforce it, we need to determine what we should apply these policies to. This is why we review the literature in order to see how the distinction of traffic can be achieved.

This very first section of the chapter will review these technologies, from the less precise ones to the ones offering the best granularity. We will thus begin with the isolation techniques based on the creation of networks or sub-networks. We will further study the solutions based on the characteristics of the traffic.

2.1.1 Network Isolation

Historically, the first use of grids and cloud computing has been made by important companies willing to reduce their costs and increase their computational capabilities by pooling their resources in remote data centers. Due to the evolution of technology in both hardware capability and virtualization techniques, this model has become more and more interesting from a financial point of view [13].

Incidentally, the cloud computing has become a business model in itself, in which a company (the cloud provider) would create remote data centers in order to lease its resources to another company. In this model, several companies, or tenants, could use the remote data centers simultaneously.

In order to protect the tenants assets, their data must be separated from one another, both on a physical server and on the communication channels. We will see in the forthcoming sections what measures have been put in place in order to do so.

Divide et impera - Physical Isolation

Some companies like government affiliated companies are handling very high sensitive data and the commercial offers of the early years of the cloud did not meet their requirements as these offers did not provide much guaranteed security. However, cloud providers then began to look more closely on how to provide a highly secured environment as they realized it consisted in a new market. In this context, new offers were quick to appear [3, 5].

In order to provide such high security levels to a tenant, one simple and yet efficient concept has emerged: using dedicated appliances in order to literally isolate the tenants from one another [2, 4]. The cloud provider assigns specific hardware to each tenant who requires the highest level of security. Therefore, the tenant's VMs are running on dedicated servers, which are physically isolated from the other tenants.

Such a technique obviously provides guaranteed isolation between the different tenants' networks and traffic.

Indeed, this physical separation prevents one tenant's data to reach one of the other tenants' machines. It is then impossible for a malicious tenant to try and capture data from another tenant by eavesdropping on the network. Furthermore, data paths are totally separated from one another as they run through different appliances, which makes impossible any disruption made by another VM on the network, such as an Identity Spoofing attack or simply a Denial-of-Service attack.

However, the use of such an architecture in a context where the network may be composed of hundreds of thousands of nodes raises many questions. To begin with, the dedicated hardware, even though it may be inactive at a given time, has to be monopolized for the tenants who need the highest security. In other words, the pooling of resources would be limited which may lead to an oversized network. Indeed, the dedicated servers could have hosted other tenants' Virtual Machines. However, these VMs would have to be located on other servers, contributing to the waste of resources, as the servers would not be loaded at their maximal capacity. Furthermore, as the use of the cloud would grow, we can easily imagine that the expansion of such a rigid architecture would become harder and harder to maintain and configure. Such a solution would cost even more as the cloud would grow, which goes against one of the reason why the cloud is widely used: reducing the waste of resources and identifying economies of scale.

For instance, we could imagine several companies with high security standards migrating to the cloud. It is unlikely that for each company, their VMs' load would be exactly the capacity of a whole number of physical servers. Therefore, each unused capacity portion of the servers would add up, leading to a bigger waste of resources. As the number of tenants would grow, the waste of resources would also grow.

Providing isolation in this manner leads to a very rigid architecture where the different tenants networks are partitioned and isolated from each other. In the case where the management of this architecture would be administrated by the cloud provider, the multiplicity of networks, middleboxes as well as routing policies regarding middlebox sequences traversal configurations would be more than puzzling. To date, no management architecture allows to dynamically administer the network in such a fashion.

Lastly, one major drawback of rigid architectures in the cloud is the migrating behavior of the nodes.

Thus, going back to the example of a small company migrating to the cloud and using dedicated servers, the cloud provider could need to migrate nodes in order to replace hardware for example. However, the migration of these nodes has to be done to other dedicated

hardware in order to preserve security. Furthermore, the migration of a node inside a tenant's network may require a routing configuration in order to maintain the middlbox traversal. Once again, no architecture allows to automatically reconfigure the cloud automatically upon these migrations, and a manual reconfiguration is simply inconceivable in networks as big as the clouds' ones.

As a conclusion, such a segmentation is efficient on the one hand, as it provides complete isolation between the different tenants while making possible the configuration and enforcement of security policies in each of these networks. However, on the other hand, the outcome of the application of that method would be a very rigid architecture where resources are duplicated instead of being pooled. This would lead to an oversized network which is problematic in a context where there are potentially hundreds of thousands VMs across the data center.

These problems would lead to the decrease of the economies of scale and to a waste or resources, hence increasing the cost of this solution. The cloud computing is a fast-evolving field and such an architecture is not likely to be put in place as the evolution of cloud management software and the virtualization technologies are now more mature as they were at the cloud premises. The scalability problem of this solution makes it unsuitable for the cloud today.

Segmentation by VLAN - Virtual Isolation

Physical segmentation is not the only way to separate several networks from one another. Indeed, the Virtual Local Area Network (VLAN) technology, as defined in the IEEE 802.1Q protocol, makes it possible to create several virtual networks independently to the network topology. A VLAN consists of a group of hosts behaving as if they were belonging to a single network segment, regardless of their physical location.

The VLAN technology is traditionally used to configure and administrate networks in enterprises as well as in the research field, as they provide the ability to easily configure and isolate a group of hosts in a virtual network. Their widespread utilization makes them inescapable in network management nowadays, and it is not rare to find them in recently proposed cloud architecture [19, 9].

In this section, we will discuss the use of VLANs in order to create a secure isolation in a cloud network. First, we will review the use of classic VLAN, where VLAN membership is defined by port or MAC address. Then, in the second part of the section, we will focus on VLAN stacking, or VLAN Q-in-Q, which extends the 802.1Q protocol.

Classic VLAN

The development of the Internet during the past decades has led to bigger and bigger Local Area Network (LAN). The switched networks, although suited for most networks, are problematic when the network is too big. Indeed, a broadcast domain too important leads to an unstable network, as flooding the network will consume a lot of bandwidth. Furthermore, another drawback of these networks are the fact that all users can see all devices in the network.

This situation has led to the emergence of VLANs, which virtually create several networks on one topology. There are two main ways to configure VLANs. First of all, assuming they support VLAN technology, each switch of the network has a table linking the VLAN IDs to its ports. By linking the switches accordingly, one can create several networks which are a sub-part of the original broadcast domain. Lastly, the VLAN membership can be based on the station's MAC addresses. That method is more practical as it is independent from the node's position in the network.

However, using solely VLANs to isolate the network has some major limitations. Indeed, when we have a closer look at the VLAN header 2.1, we can see that the the VLAN ID field is a 12-bit tag, as defined in the Institute of Electrical and Electronics Engineers (IEEE) 802.1Q standard [8], which implies a limit of 4096 distinct coexisting networks. This number is too low when we take in consideration that a major adoption of the cloud would create immense data centers.

Furthermore, if several policies have to be put in place for a tenant, the number of required sub-networks would be even greater. Thus, VLAN technology cannot be used as it is a limitation to the scalability of the cloud model.

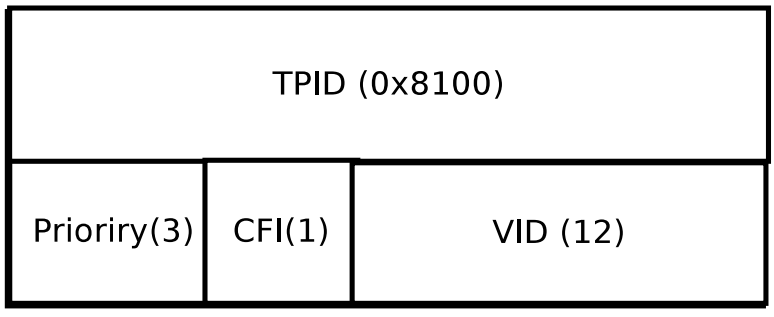

Figure 2.1 VLAN header

VLAN Q-in-Q

As the Internet has evolved, the need to use VLANs in order to tunnel packets that were

already tagged has appeared, as a result to ever bigger LANs. This is why the IEEE 802.1ad standard includes the definition of VLAN Q-in-Q, also called double tagging. As its name suggests, double tagging consists in using two tags instead of one [6]. This allows the service providers and the end user to concurrently use VLANs in a heterogeneous network.

As a consequence, The double tagging theoretically increases the numbers of possible networks from 4096 to 16777216 different networks, even though it was not the main goal of this technology. However, classic routers do not have the capability to analyze these two tags as a couple, so it does not help the scalability problem of the cloud. Furthermore, the routers still have to be configured manually, which is another issue in an environment which needs to be automated.

Conclusion on VLANs

We have seen through this section the drawbacks of using VLAN technology in today's and tomorrow's clouds. Due to its wide spread use among enterprises and researchers, it is normal that they were used to secure growing data centers. However, the cloud is ever growing and has now entered a phase where the main requirement is scalability, which VLAN fails to provide.

Furthermore, the economies of scale are requiring a simple and automatic management system. Once again, the VLAN does not qualify as a potential technology because of its manual configuration on the routers along the paths. Consequently, dividing the traffic using the classic Virtual Local Area Network technology is an inadequate solution for the cloud.

Mixed Segmentation

As neither the physical nor the virtual separation are satisfying the cloud's requirements, it is time to evaluate the combination of those two.

A recent cloud architecture proposal [15] is using VLANs in addition to a particular physical topology in order to isolate tenant networks from one another. The whole cloud network is indeed divided in several parts: one core domain, in which all traffic arrives to the data center, mainly composed of routers, and several edge domains, where the tenants VMs are located.

The division in multiple domain compensates the scalability problem of the VLAN technology, as the number of nodes are then reduced and can fit the maximum number of VLANs.

In the case of a tenant's nodes being in several edge domains, another identifier is added to the tenant's traffic: the cnet ID. Furthermore, in each edge domain, each network is isolated by VLAN. When a tenant's VMs are located in more than one edge domains, the cnet id is used in combination with the VLAN id to make the connection between the virtual networks.

Moreover, critical competitors can be isolated from one another as they can be set in different edge domains. Due to its architecture, the scalability will not be as impaired as it is when the physical isolation is used.

This work is a good example of the difficulty of using VLANs to isolate and administrate tenants' networks inside a data center. The scalability issue the VLAN technology has led here to the need of separating the network in several edge domains as well as adding a new identifier.

Providing a scalable architecture isolating the tenants networks from one another is therefore possible. However, it is a rather complex solution which is not automated, and on top of which we wish to add the automatic enforcement of security policies.

2.1.2 Flow-based Segmentation Architectures

In this section, we will analyze the segmentation techniques that are not based on a particular technology, but all have one thing in common: they use the flow characteristics to differentiate tenant traffic and thus enforcing security policies. These characteristics could be the destination IP address, the source MAC address, the protocol type, or any variable which could characterize the flow of packets.

We further subdivide these technologies in two categories, which we call Gateway-based techniques and Network infrastructure-based techniques. The first ones have dedicated forwarding elements which act as gateways, and that are responsible for detecting the flows and enforcing policies. In the second category, all routing elements can potentially act as policy enforcement devices as well as forwarding elements.

Gateway-based Architectures

We analyze here the solutions where the traffic goes through forwarding and policy applying elements. These elements are responsible for detecting, redirecting the flows and applying security policies. Figure 2.2 simply illustrates how the traffic can be differentiated by the forwarding elements placed between the core domain and the other domains. In this figure, these elements are represented in blue.

The solution studied in the previous section [15] is designed in the same way. In order to achieve the flow separation, the forwarding elements lies at the frontier of what they call Edge Domains and the Core Domain.

A central controller has the mapping of the running VMs and can then distribute them into the Edge Domains, as the Core Domain will mainly act as a transit domain. The forwarding elements lying at the frontiers of the domains are connected to the controller

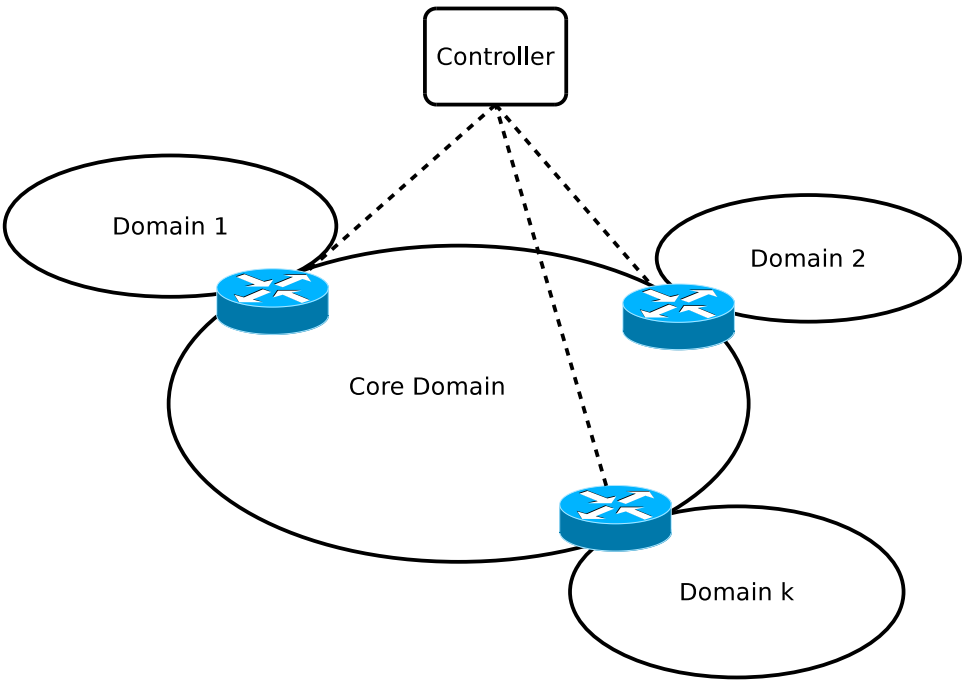

Figure 2.2 Gateway-based segmentation

which hold the policy databases.

Therefore, when a tenant's packet is detected at the frontier, the according routing rules are set, based on the security policies. Furthermore, particular rules can be set based on the identity of the emitter or receiver of the packet. The application of middlebox-based security policies will be analyzed in section 2.2.2.

In other solutions found in the literature [19], the gateway-based model differs slightly from what is described above. Special forwarding elements named *pswitches* are spread across the network. As they process a packet, the pswitch communicates with the centralized policy controller in order to know how to steer the packet across the network.

The pswitches can match the flows based on the five tuple (source IP, destination IP, source port, destination port and protocol type).

Gateway-based techniques present a drawback as the gateways act as bottlenecks for the traffic, which is bad for the scalability of the solution, considering that the data centers host large networks that generate an important amount of traffic.

Network Infrastructure-based Techniques

In the previous section, the traffic was identified by gateways acting like sentinels across the network. In this subsection, we analyze solutions where the flows are identified and processed by all the elements of the whole network infrastructure.

During the past few years, important advances have been made that have completely modified the way we approach networking. In classical routing, the high level decisions (the control plane), such as deciding which port to send the packet out to, as well as the packet forwarding itself (the data path) occurred on the same appliance. However, recent research tend to take the control path out of the switches and centralize it [14, 12, 22].

Therefore, the logic on which the decisions are made is located in the centralized controller, as the switches are simple forwarding elements devoid of any logic.

2.1.3 Openflow

One particular Software-Defined Network protocol [22] draws the attention of the whole community. In the OF protocol, the data plane and the control plane are totally separated. As a packet is received by a switch, it forwards the packet to the controller which analyzes the packet and decides how to process the packet. Then, the controller adds an entry to the flow table of the switch. The flow entry contains the requisite information in order to match the packet as well as the action to apply to the matching packets. As the switch receives a second packet belonging to known flow, there is no need to communicate with the controller and the packet is immediately processed.

Consequently, several logical networks can be defined by specific characteristics and will then be isolated from each other. The precision of the flow-definition depends on the implementation. The first generation of OpenFlow (OF) switches, the flow can be matched based on a 10-tuple [11]. The matching fields are shown on figure 2.3. In such an architecture, the granularity of the isolation is as good as the precision with which the flows are differentiated.

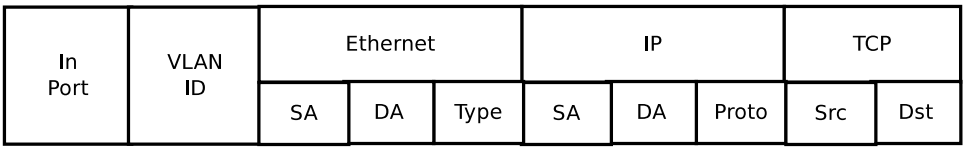

Figure 2.3 The header fields matched in the OF protocol

Using this segmentation technique allows to have a centralized controller which orchestrates the traffic in order to apply the right policies [17, 23]. It is also possible to have several controllers, each one of them directing a slice of the network [27].

Recent research [21] have worked on creating and distributing a Network Information Base which contains the information regarding the nodes and the network topology. By created a distributed model, they ensure a suitable scalability regarding the cloud requirements. Furthermore, it is possible replicate the controller in order to reduce its load and improve its performance.

As a conclusion, a flow-based segmentation of the traffic provides a good isolation, scalability and automatism. The logic of the routing is concentrated in the central controller which automatically pushes the rules down the switches based on a wider logic. Software-Defined Network seems at this point the most relevant technology in order to match the traffic whom we further want to apply the security policies to.

2.2 Security Policy Enforcement

As an enterprise migrates to the cloud, they primarily want to reduce their costs while benefiting of a better and more secure solution [20]. In an on-site installation, companies usually benefit from security policies set and maintained by the IT department.

This section is divided in two parts. We will first review the literature on how the security policies are defined in recent cloud architecture proposals. Then, we will review the various possible ways of achieving the enforcement of these policies.

2.2.1 Security Policy Definition

Our work focuses on the enforcement of security policies at the network level. In this section, we will consider the security policies as they are defined in numerous recent work. We believe that it will help us define what we want to achieve in our solution.

First of all, we will consider security policies as routing rules, or more precisely, allowed and forbidden actions. We will see what they mean as well as how they relate to tenant isolation.

In the second part, we will consider the security policies as implemented by a chain of middleboxes. We will analyze how these chains of middleboxes are defined and how the security policies are applied.

What do the policies target

Many people call their mother on the phone. Sometimes, we chat with a stranger at the bakery shop. Yet, it would be awkward for the stranger from the bakery shop to call your mother. This is because in life, as well as in networking, there must be rules set in order to allow or deny certain communications.

These rules are important in networking as they ensure the security of sensible data by restricting access to some machines. However, in order to apply these rules, we need to determine a set of constraints to match. These constraints however depends on the type of traffic isolation put in place.

In recent work [17, 18], research projects aim to provide a declarative policy language for managing the enterprise networks (FML/FSL). As it is done in the OF technology, the matching of the flows is made on a tuple representing multiple characteristics of the flow, such as target and source address. The matching fields defining a flow in the FML are shown on figure 2.4.

Property	**Description**
u_s and u_t	Source and Target Users
h_s and h_t	Source and Target Hosts
a_s and a_t	Source and Target Access Points
prot	Protocol
request	Whether or not flow is a request.

Figure 2.4 Characteristics of a flow [17]

The difference from the other works relies on the request property. Indeed, in this language, the authors deem that the policy engine must be capable of determining whether a flow is a request or a response to a request.

This research provides a good abstraction of the possible security policies that can be applied. However, this new declarative language is done in order to simplify the management of the network. Therefore, it must suffer from an over-complicated form when it has to be used by the tenants on a day-to-day management of their cloud.

Bellessa et al. propose NetOdessa [23] in order to tackle the error-prone configuration of explicitly writing rules about individual hosts. They build on previous research [10] on a distributed host-level dynamic policy monitoring system, in order to extend it to the network layer.

The network manager can write basic policies about groups of hosts which are then translated and enforced at the network layer. The host agents then monitors the behavior on the nodes in order to detect whether the policies are consistent or not. The data gathered is used to refine the enforced policies by taking actions in order to rectify the application of the policy.

The main focus of the work [23] is the detection of violations. In its design, the NetOdessa proposal allows the definition of a violation as a policy. The given example is the one of the denial of the use of SSH by anonymous hosts. As an anonymous host is trying to communicate through SSH, the host-level monitoring system will detect the violation and prevent the host to do so.

Another advantage provided by this solution is to detect the compromised nodes without

having to shut down the whole group of hosts the compromised one belongs to. The dynamism brought along by the host-level monitoring system allows the detection of a suspicious and potentially malicious change on a node, and then actions can be taken at the network level. This dual surveillance is what is particular of this work, compared to the other research on the subject.

We have already evoked the paper [15] on secure cloud computing several times. In this architecture, where the network is divided in a core domain, on which are bound edge domains, the forwarding elements placed on the network frontiers are distinguishing the flows based on their membership to a tenant network.

The authors decided to apply policies based on the identity of the destination of the packets. The forwarding elements then act as a firewall which can block and even redirect the packets based on the identity of the destination machine.

A problem occurs when two VMs of one tenant communicate with each other and security is required. Indeed, if the two VMs are in the same VLAN, the traffic will not go through a forwarding element and thus no policy can be applied.

The solution requires for VMs to be placed on separated VLANs when a security policy must be applied. However, what would happen if all communications of the VMs must be run through a middlebox? Using one VLAN per machine does not seem appropriate in this case.

We have reviewed several work in this section. Some are more focused on the policy definition [17, 18] while others focus on the ease of network management and security [23, 15]. In every case, we analyzed the strengths and weaknesses of the solution.

After having analyzed the target of the policies, we will focus on the policies themselves in the next part.

Policy Definition

Now that we know what are the target of the policies, we want to have a closer look on the policies themselves. We aim at analyzing the recent cloud architecture proposals providing security in order to have a better understanding on the use of middleboxes.

We begin by continuing and finishing our analysis on the paper [15] where the policies are applied per user. The policies can consist in filtering the traffic, applying Quality of Service treatment or route the traffic through middleboxes.

There is yet no evocation of the traversal of a sequence of middleboxes.

The distributed host-level dynamic policy monitoring system proposal [23] goes one step further in the elaboration of policies.

The policy is here defined by several rules, each rule being a combination of a condition

and an action. As the condition is satisfied, the action is applied. The conditions are set of statements written using the Resource Description Framework [1]. An action to be taken is a change in the network triggered by the inference engine. An action can be the set of routing rules as well as dropping a flow, or routing a flow through a middlebox.

Even thought it takes in consideration the traversal of middleboxes in their design, it is not specified how the routing itself through the middlebox is going to be achieved. Furthermore, there is no evocation of the traversal of a sequence of middleboxes, which we believe is key in order to achieve a level of security comparable to the one offered in an on-site installation of an enterprise network.

FML/FSL allow the network manager to define groups of hosts and configure policies.

The constraints that can be applied to these groups of host are of five types: *allow, deny, waypoint, avoid* and *ratelimit*. The *allow* constraint, as its name suggests, is used in order to explicitly define authorized communications. The *deny* constraints, almost as obviously as the latter, means that the flows associated to this constraint in the policy will be denied. *Waypoint* requires a flow to pass through a particular node of the network and "avoid" will forbid a flow from passing through a particular node. Finally, the keyword *ratelimit* will allow for the network manager to set the maximum bandwidth of a link.

This policy configuration allows for the tenant to easily define simple rules in a similar fashion as what is done in an on-site enterprise network, where a member of a department D1 can reach the department D2 but must be strictly prevented from communicating with D3, where the sensible information lies.

This paper is the only one to specifically address the problem of middlebox traversal. The centralized controller holds the policy database which contains the sequences of middleboxes to be traversed. A policy is of the form : *Location, Traffic Selector, Sequence*. The *sequence* being the sequence of middleboxes.

The *Start Location* defines the physical location from which the packets are emitting. The *Traffic Selector* will use the five-tuple in order to define which flows the security policies must be applied to.

This paper is interesting as it is the only one considering the enforcement of a sequence of middleboxes in a cloud network.

2.2.2 Policy Enforcement

The goal of our work is to provide a secure solution to the tenants as they migrate to the cloud. From the beginning, we identify the security policy as a sequence of middleboxes. Indeed, we want to provide the same level of security in the cloud as in the enterprises, as the middleboxes are widely implanted.

We have seen in the previous section that pretty much all the solutions support the middlebox traversal in their architecture, even though they don't allow the traversal of a middlebox sequence.

In this section, we go one step further by analyzing how the policies described in the previous section can be enforced at the network level.

Rigid Architecture

We begin this section by analyzing how the policies can be enforced when we have a rigid architecture, and what are the problems which can occur.

Topological segmentation can physically separate several networks from one another.

Enforcing security policies in this context is possible. Indeed, the middleboxes that have to be traversed can be located in the tenant's network before setting the routes accordingly. Each tenant's traffic is then going through the right middleboxes and the traffic is secured.

However, for each tenant and each network, there has to be an instance of each middlebox, or at least of each middlebox that has to be traversed. This duplication of middlebox is severely impairing the scalability of this model. Furthermore, the load of the middleboxes is not optimized as they would be dedicated to a particular tenant whether his traffic is important or not.

For example, we can imagine a small company with high security requirements that is using the cloud to run its applications. Several middleboxes would then be monopolized for this tenant, even though these middleboxes are far from being used at their full capacity in terms of traffic load. Furthermore, the expansion of the data center would once again contribute to an important waste of resources.

Some cloud architectures proposals suffer from a rigid architecture [15]. In such cases, the network is partitioned in smaller domains, and the forwarding elements enabling the enforcement of the security policies lies at the frontiers of the networks.

In this situation, the middleboxes have to be connected directly to the forwarding elements. The problem here is the rigidity of the structure. All middleboxes are concentrated at the frontier, and all secure traffic must also run through these gateways. This has for effect the creation of bottlenecks in the network, imputing the scalability of the system.

The paper [19] also suffers from this drawback, but in a lesser extent, this is why we decide to review it in the next section.

Off-path Middleboxes

In this section, we consider the situation where the middlebox can be relatively anywhere in the network topology. We will see how the presented solutions achieve to route the packets from a source, to a destination, and throughout a sequence of middleboxes.

FML, proposed by Hinrichs et al. [17] provides the capability to define policies including *waypoints*. The *waypoint* constraint will force a flow to go through a particular node before reaching its destination.

When the packet is detected, the correspondent security policy is retrieved. Then, the policy enforcement engine will calculate the path from the source to the destination, respecting the constraints of the policy. In the case of a *waypoint*, the route will be calculated considering the traversal of the required *waypoints*. Then, the route will be enforced at the network level.

Thereby, this solution can be used in order to enforce a security policy in the form of the traversal of a middlebox. It is nonetheless not explicit if the algorithm allows the routing nor the traversal of a sequence of middeleboxes, but even if it does not, we believe that the capability could be added by implementing minor changes to the algorithm.

Another drawback is more obvious when we consider the migration of nodes. Indeed, the *waypoint* defines a particular node to be traversed whereas a security policy requires the traversal, for instance, of a firewall. However, it may happen that there are several firewall instances running inside the data center. Moreover, as a Virtual Machine migrates, the routing throughout the same firewall as the one traversed prior to the migration is likely to be problematic, as the traversal of another firewall instances could potentially be more efficient.

Due to the fact of the lack of an enforcement mechanism and the application of sequences of middleboxes, as well as the low tolerance to node migration, we believe this solution can not be used for our purpose, even though it can be a source of inspiration.

One of the recent works on cloud architecture [19] proposes an architecture in order to dynamically apply security policies, and to address the limitations of current middlebox deployment mechanisms. In this model, the policy is defined as a sequence of middleboxes.

The main goal is to enforce security policies to each different traffic. The forwarding elements, which are the gateways, are used to steer the traffic towards the right middleboxes by analyzing and encapsulating the packets. The gateways are spread across the network, only a limited number of routers act as gateways.

The traffic is encapsulated in Ethernet packets in order to force its path through the right

middleboxes. Indeed, the middleboxes are plugged directly into the pswitches, those modified switches placed across the network. As a packet arrives to the network, it is detected by a pswitch which encapsulates the frame into another Ethernet frame addressed to the pswitch in which the middlebox is plugged.

Thereby, the packet is routed through the network as any other packet, using no particular mechanism, as the destination address of the frame is no longer the address of the destination machine, it is the address of the next middlebox ingress switch.

Once the packet has reached the middlebox ingress switch, the encapsulation is no longer needed. Furthermore, many middleboxes use the header information in order to analyze a packet [25, 16]. This is why the packet is decapsulated before being sent to the middlebox, and in case there is another middlebox towards which the packet must be sent, a new encapsulation is operated.

This model used to present an advantage as it enabled the use of only a limited amount of dedicated routers as gateways, which is interesting in the context of making an existing network more secure with minimal changes made to the infrastructure. However, our study focuses on what should be the state of the art solution to meet the challenges that cloud operators have to face, even if it means building the architecture from scratch.

Furthermore, The repeated encapsulation and decapsulation of packets lead to a waste of resources, which is critical in a cloud environment. In addition, the migration of nodes has not been considered.

We can see that across the pswitches, the routing is done based on the next hop. We consider two VMs, VM1 and VM2. When VM1 migrates and starts emitting packets again, the routing based on the next hop is problematic in case of overlapping routes between VM1 and VM2, before and after migration. Indeed, the rules in the pswitch will still match the source and destination fields of the two VMs, even though their location has changed.

The policies do not remain consistent despite the migration of nodes, as the problem is addressed in a static context.

2.3 Conclusion

In this chapter, we studied the literature in order to realize what has been done in Cloud Computing in order to enforce security policies at the network level. Particularly, we considered the criteria of scalability and automatism in a context where the network is shared by multiple tenants and the migration of nodes is omnipresent.

We first considered the different existing solutions in order to differentiate the traffic between the different tenants, as the isolation between tenants is the basis of a secure network.

There are many solutions allowing traffic separation. However, the precision of these solutions varies greatly, as the isolation can be made from a complete separation of physical network [2, 4] to the flow-by-flow separation provided by the matching of shared characteristics of the packets [22, 19, 17].

The recent breakthroughs in Software-Defined Networking have let us to believe that the most suitable technology for our project today is the use of the OpenFlow technology [22, 11] as it provides an isolation per flow as well as a automatic centralized control of the network.

We then analyzed the way security policies are defined and enforced. The security policies are defined in several ways. Some solutions focus on the isolation only [24] or the routing rules and their consistency [23], whereas other architectures put more emphasis on the traversal of middleboxes [19, 15].

Furthermore, we have studied the mechanisms allowing the traversal of middleboxes. Solutions providing a rigid architecture where the middleboxes are topologically constrained [15] is not satisfying for us, as those concentrations of middleboxes can act as bottlenecks in the network.

Only a few solutions [19, 17] allow the routing throughout middleboxes whose location is not constrained, to a certain extent. However, these solutions either fail to provide the traversal of sequences of middleboxes [17] or their policies do not remain consistent through the migration of nodes [19, 17].

None of these solutions are suitable to our requirements, as we consider the traversal of sequences of middleboxes in a context where the migration of nodes is not unusual. Yet, they pave the way towards the security of the cloud networks.

Through this literature review, we studied the different ways possible in order to achieve the enforcement of security policies in a multi-tenant cloud network. We are now able to build on this knowledge in order to build a solution of our own in the course of the forthcoming chapters.

CHAPTER 3

DESIGNING AN ELASTIC ENFORCEMENT LAYER

The previous chapter provided a literature review on the techniques used nowadays in order to secure the tenants networks by implementing security policies. Although these solutions are tackling that particular problems, we believe they are not suitable for today's multi-tenant cloud networks. Indeed, these solutions are not remaining consistent through node migrations. Furthermore, many of the current solutions show scalability issues.

In this chapter, we build on what we have learned in the previous chapter in order to develop our own solution. Innovation means enhancing a product, increasing its performance or providing new or ameliorated services. This is what we try to do in this project. We want to give the cloud provider the possibility to dynamically enforce security policies in a multi-tenant cloud network.

In order to do so, we will first define the use cases in which we want to provide a solution. It will help us define the behavior we want our solution to adopt.

Then, we will analyze and refine our requirements regarding the solution we plan on developing, based on the use cases as well as on the literature review. We will analyze each aspect of the problem that we want to address, and also precise the scope of our study. Next, we will precise how we plan on meeting each requirement, and which technology we plan on using.

Lastly, we will present the complete flowchart of the algorithm of our solution, which will include and summarize the proposed solutions for each requirement. We will also go through the use cases once again in order to see the behavior of our algorithm.

3.1 Use Cases

Every technology is built in order to answer a particular problem or at least to facilitate the management of a practical situation. This is why we believe that the analysis of practical use cases is key in order to define the most relevant answer to provide.

In this section, we will study three use cases, which are key in order to define our requirements. We believe they reflect the needs of the tenants when they move their networks and applications to the cloud.

When a cloud provider is running a tenant's VMs, those Virtual Machines could be located anywhere inside the data center, on any physical server. Based on the type of machine, or

based on the type of traffic which is emitted, the traffic from one VM to another is submitted to certain security requirements, defined by the tenant. These requirements correspond a chain of middleboxes that have to be traversed. Therefore, the route between two VMs has to be redirected in order to go through these particular middleboxes.

We will name “ingress switch” and “egress switch” the switches traversed right before and after the middlebox. In our case, we will consider the ingress and egress switches to be the same.

The 3.1 provides a simple view of the abstraction of the Cloud which could be presented to the tenants. Indeed, our goal is to dynamically apply the security policies, regardless of the location of the different VMs inside the data center.

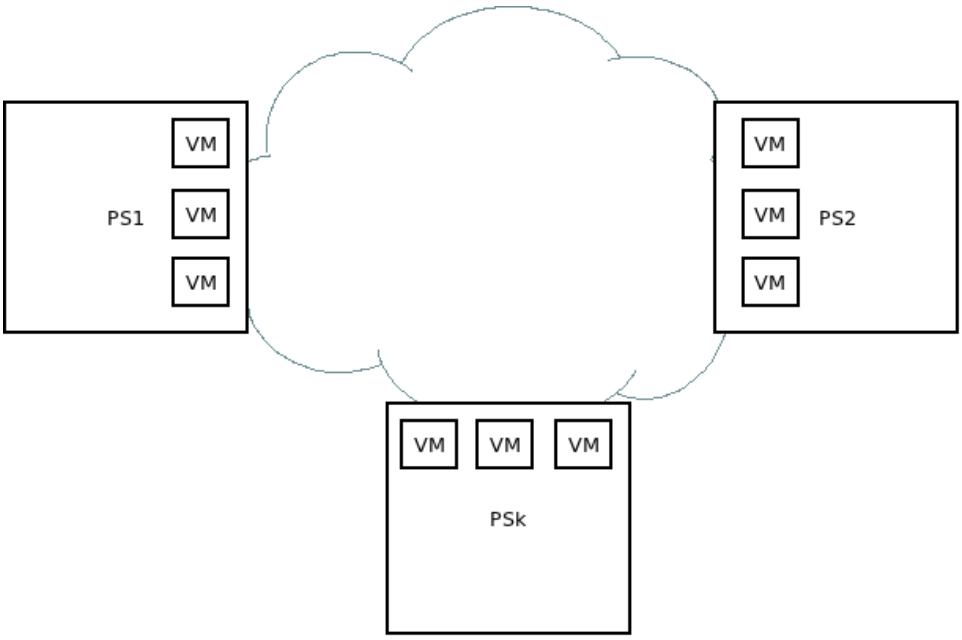

Figure 3.1 Communication between two VMs

3.1.1 The Basic Use Case

This first use case is illustrated on figure 3.2.

Two VMs located in the network will communicate. As soon as they do, adequate routing rules have to be set for the traffic to go through the right middlebox instances. The selection of the instances could be based on the identity of the VMs or the type of traffic emitted.

More precisely, we consider two VMs, VM1 and VM2 , located on different physical servers, respectively PS1 and PS2. Several middleboxes are located across the network. In this particular example, we consider three middlebox instances : M1, M2 and M3.

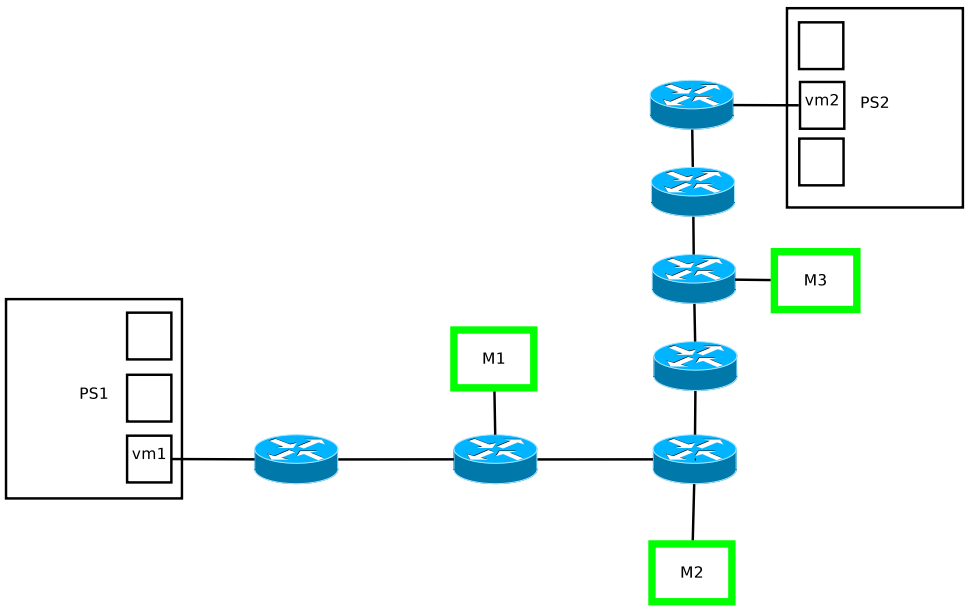

Figure 3.2 Basic use case

Scenario 1

VM1 is running an application which needs to communicate with VM2. As the first packet is emitted, the first step of the process is to identify the flow by its characteristics.

Once the flow is identified, it is translated into a middlebox sequence to be traversed. This security chain has to be taken in consideration as the path to take is determined. Let's assume the middlebox chain is M1-M2-M3.

Furthermore, the routing rules have to be set, in order to allow the traffic to go throughout the proper devices. In our case, it means going from VM1 to M1's ingress switch, then through M1 to M1's egress switch. The packets are then routed to M2's ingress switch and similarly through M2 to M2's egress switch. The same is done to go through M3. Then, the packets are routed from M3's egress switch to VM2.

Scenario 2

Let's assume that VM1 is running another application. This application could be using a different protocol, or using different ports. The traffic of this other application could be identify as another flow.

Once this new flow is identified, it corresponds also to a security chain. This security chain could be the same as previous or could be different. Let's assume that the security chain is now different, for example, M2-M3.

The routing rules related to this new security chain also have to be put in place in the same way as previous.

Conclusion

In this very first use case, we distinguished two scenarios which would likely occur. Based on these scenarios, we can already identify several key issues that our solution will have to address.

First of all, the different flows have to be identified and linked to a particular security chain. As we have seen in the literature review, the distinction between different tenants traffic could be done by identifying the characteristics of the flow.

In this use case, we can see that it would be interesting to have an identification which could separate distinct flows even within a tenant's network, and even on a sole machine, in order to provide a particular security policy for each type of application which could be running.

Furthermore, both these scenarios could happen at the same time. We can sense that the routing rules which would have to be put in place would be relatively identical, as the security chains are quite similar (M1-M2-M3 vs. M2-M3).

Because of the size of nowadays' data centers, the multiplication of identical rules could be a hindrance to the scalability of our model. Therefore, it would be interesting to be able to aggregate these rule.

3.1.2 Reconfiguration upon Migration

The second use case is a little more complex than the first one. Two VMs are communicating in a similar way as previous. However, in a data center, the cloud provider will ask for a VM to migrate to another physical server. The migration of VMs can be quite useful, for example when a hardware have to be replaced, or in order to optimize the load of the physical servers. In this use case, we analyze what should happen to allow the security policies to remain consistent despite the node migration.

In this use case, we consider two VMs, VM1 and VM2. These two VMs are located on different physical servers, respectively PS1 and PS2. As previous, several middleboxes are located across the network. We consider four middleboxes, M1, M2, M3 and M4.

Another physical server is present in this example, PS3. In our scenarios, the VM1 will migrate from PS1 to PS3.

This use case is illustrated on the 3.3.

Scenario 1 - Part 1

VM1 is running an application which needs to communicate with VM2. In the same fashion as in the first use case, the first step will be the identification of the flow.

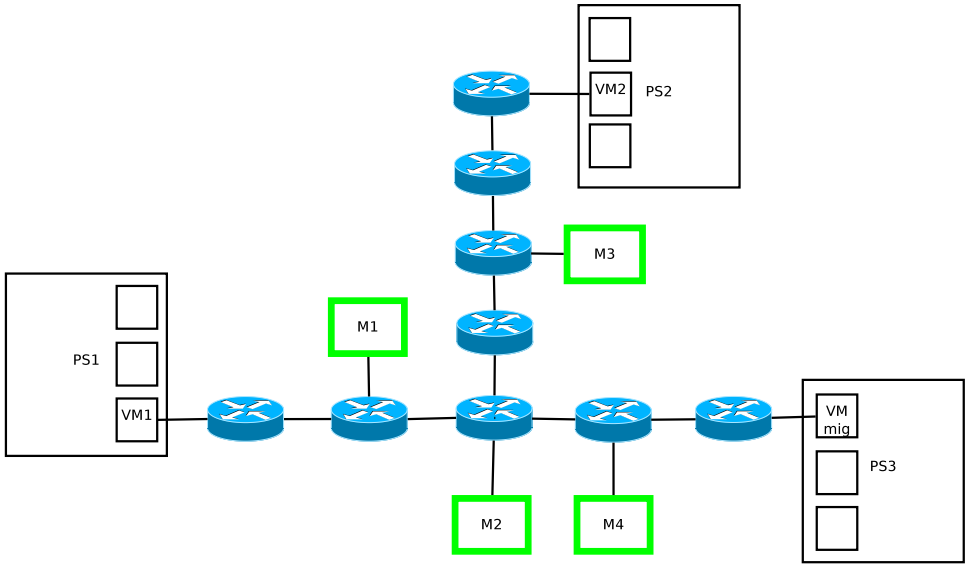

Figure 3.3 Case of a migration

Once the flow is identified, it corresponds to a middlebox chain to be traversed. Let's assume the middlebox chain is M1-M2-M3.

Finally, the routing rules have to be set at the switches level, in order to allow the traffic to go throughout the proper devices. The last step is the same as previous.

Scenario 1 - Part 2

Let's assume that VM1 is migrated from PS1 to PS3. The new occurrence of VM1 is noted VM-mig on the 3.3.

The application running on VM1 is still running on VM-mig, and after the migration, the first packet emitted will be identified in the same way as previous.

Once the flow is identified, it is mapped to a middlebox chain. Previously, this traffic corresponded to M1-M2-M3. The most basic situation would be to continue to route the packets throughout the middleboxes M1, M2, and M3. However, it may be possible that the middlebox M1 and M4 are two instances of the same type of middlebox. For instance, M1 and M4 could be both a firewall. In this particular case, it would be interesting to consider the chain M4-M2-M3 instead of M1-M2-M3.

Scenario 2 In this second scenario, we consider the migration of the destination node, instead of the source node. Concretely, it means that VM2 is migrating from PS1 to PS2, and would then be named VM-mig.

Upon a migration, the rules set which route the packets from VM1 to VM2 should be

discarded in order to allow VM2 to keep on receiving the messages coming from VM1.

Once the rules have been discarded, the next packet sent by VM1 will be identified, and the rules will be set according to the corresponding security chain, in the same way as previous.

Conclusion

One particularity of the cloud network is the node migration. This use case presents how the application of security policies could be impacted by the migration of the sender or the receiver. From these scenarios, we identify two key issue in order to build an efficient solution.

First of all, we have seen that the flow must be identified in order to apply the right security policy. Thereby, the first question raised is how to efficiently identify a node or an application's traffic, despite the location of the node in the network. As the identification of the node is key to apply a policy, the identification must be consistent, regardless the node's migration.

The second issue raised here is the consistency of existing rules after the migration of a node. Indeed, two cases are present here. First, when the sender migrates, the new rules set must be coherent with the security policies. Secondly, when the receiver is migrating, the previous routing rules must be discarded, or modified in order to allow the traffic to reach the receiver. An optimal solution would take this into account.

3.1.3 Communication between Distinct Zones

In order to the solution to be scalable, the whole network to secure or to manage is often partitioned in smaller sub-networks, each ruled by one controller. In such a scenario, it would be possible for two communicating nodes to belong to different sub-networks, or zones.

This third and last use case is another variation of the first use case. Two VMs are communicating in a similar way as previous. However, in this case, they belong in two different regions ; each region being ruled by a distinct network controller.

In this use case, we consider VM1 and VM2, located on different physical servers, respectively PS1 and PS2. In this case, these two VMs, *a fortiori* these two physical servers, are located in two different zones, identified as Zone 1 and Zone 2 on the 3.4. As previous, four middleboxes are located across the network.

Scenario

VM1 is running an application which needs to communicate with VM2. As the first packet is emitted and identified as previous.

Once the flow is identified, the middlebox chain to be traversed is determined, as previous.

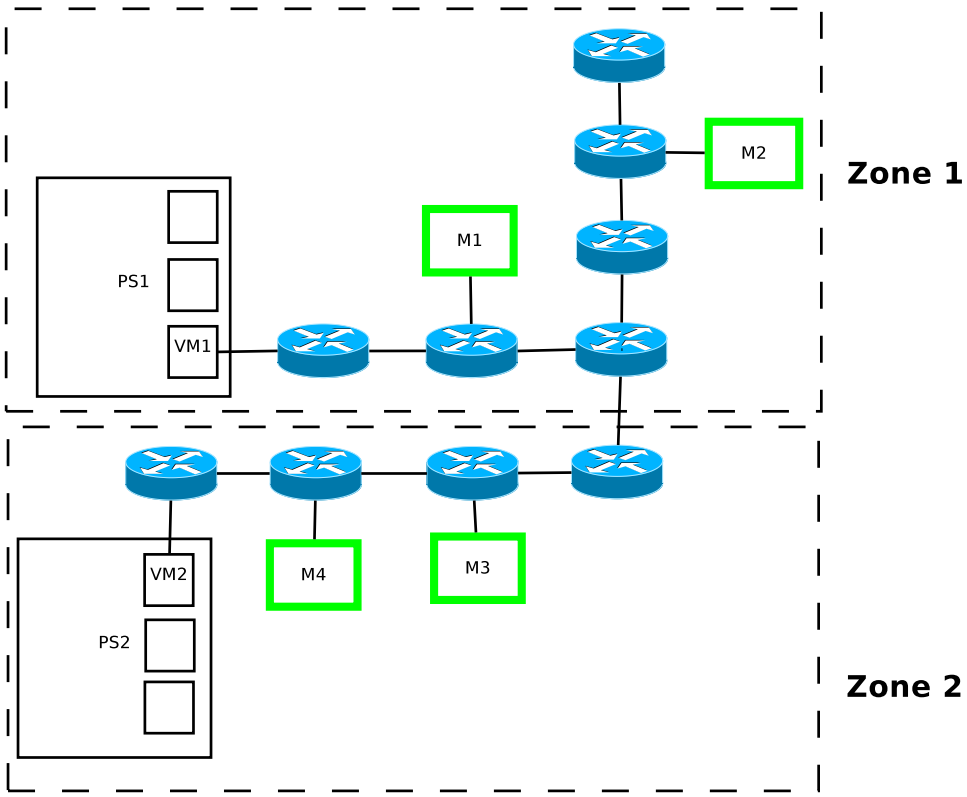

Figure 3.4 Communication between two zones

One particularity of this case is to know where to send the packet to. Indeed, the Zone 1 may not be aware of the location of VM2.

Once the Zone 1 has resolved to which gateway switch the packet needs to be sent to, it has to decide throughout which middleboxes the packet must be sent through. This is problematic as the route from VM1 to VM2 go through several distinct zones.

Let's assume that M1 and M3 are middleboxes of the same type, for instance, firewalls. Furthermore, let's assume M2 and M4 are also of the same type, for instance, Deep Packet Inspection (DPI). Finally, let's assume that the security chain corresponds to the traversal of a firewall and a DPI.

We can see that in this scenario, there are several possibilities. First, the security chain could be completely realized in Zone 1, by passing through M1 then M2, before being routed towards Zone 2. Then, another possibility is to apply the middlebox traversal completely in Zone 2, by traversing M3 and M4. Finally, it would also be possible to traverse one middlebox in each zone by going through M1 in Zone 1 and M4 in Zone 2.

Conclusion

Due to the large size of Cloud networks, the partitioning of a big network in smaller entities

has often been done. In this use case, we were interested in how the enforcement of security policies could be done in a context where the sender and the receiver are located in distinct zones. From this use case, we highlighted several issues which need to be addressed.

The first issue highlighted is the one regarding the routing of packets from one zone to another. Indeed, when the destination is not reachable directly, there must be a mechanism in order to determine the gateway to send the packet to.

Moreover, once the flow is identified and the destination is known, the routing rules must be set according to the middlebox instances to traverse. Indeed, as we have seen in the last example, the policy enforcement can be made totally in a zone or be mixed between the two zones.

Finally, there is another issue regarding the destination Zone. Indeed, when a packet arrives, it is of identified, and the corresponding middlebox chain is retrieved. In this context, how can the second zone determine which portion of the middlebox chain has already been enforced in the first zone?

3.1.4 Conclusion on the Use Cases

In this section, we analyzed three different use cases in order to visualize the type of situations which could occur in a cloud network where a policy enforcement mechanism is in place. From these situations, we highlighted several issue in order to have a better understanding of what our solution is designed for.

First of all, the flow identification will be a key aspect of the mechanism, as it has to be done in a consistent manner, despite the node migration.

Second of all, the node migration leads to the search of an enforcement mechanism which respects the consistency of the security policies.

The scalability of the solution is a crucial requirement in the cloud. We will analyze how we should aggregate the flows in order to reduce the number of routing rules inside the data center.

Finally, as the data center may be partitioned, the enforcement mechanism must support the application of security policies distributed on different zones ruled by different controllers.

3.2 Security Requirements

In this section, we will the define the security requirements of our model. Those are the properties we want our architecture to possess in order to secure the multi-tenant cloud network.

As we described in the introduction, our problem only takes in consideration the attack of a tenant by another tenant, or by malicious nodes within its network.

3.2.1 Isolation

Traffic isolation is the first requirement which was identified from the beginning, and which is key in order to build a secure network, as it prevent a tenant to be threatened by another tenant. Furthermore, We will consider only the traffic isolation at the network level. The isolation at the hypevisor level as well as any other type of isolation will be considered to be out of scope in our project.

3.2.2 Applying Security Policies

In this context, security policy means a chain of middleboxes which need to be traversed.

For a security chain, a particular middlebox type can be present only once in the chain. For example, if a security chain is composed of three elements, and the first one is of the firewall type, then neither the second nor the third could be a firewall.

Furthermore, our solution must enforce the totality of the security policy, not less, not more. Indeed, when a security policy is set, we will not allow for the traffic to traverse more or less middleboxes than required. The sequence in which the middleboxes are traversed is also very important and must not be modified. When the entire security policy can not be applied, the packet will be drop by default.

At the network level, our solution aims solely to enforce the route between the source and the destination, throughout the middlebox instances selected.

Regarding the middlebox instance selection mechanism, we decide not to set any constraints. We leave it out of scope albeit we believe it could be further investigated in future researches.

3.2.3 Tolerance to Node Migration

Thid problem is particular to the cloud environment. As the nodes migrate inside the data center, the security policies have to remain consistent. This is an important stake, as we would not want for the security policies to become obsolete at each migration of a node.

As we highlighted in the section 3.1, the migration could concern either the sender of the receiver of a communication. In our work, we will only consider the sender's migration, and focus on the reconfiguration of the network.

Indeed, as the receiver would migrate, we would need be to discard the rules. In order to discard the rules, we must first detect the migration. For instance, this migration could be

detected either by a signal send by the cloud operating system or by a modification on the network's topology. We believe it is better to leave this concern out of scope.

3.2.4 Scalability

The use of computers and networks has become dominant in our modern society. The cloud is, more than others, very sensible to the scalability of the solutions in place, due to the immense size of the data centers.

As we have seen in the 3.1, several similar routing rules could be set at the forwarding elements level. One of our goal will be to find a way to aggregate those rules. For instance, when several flows will be routed from a middlebox to another, we will try to use a unique rule which would match all different cases, insofar as possible.

As the data center would reach a critical size, it is likely that its network will be partitioned in smaller networks, in order to achieve reasonable performance. Therefore, the sender and the receiver of a communication could be located in two different zones, ruled by two different controllers.

The solution must allow for the middleboxes to be traversed in the same zone or in different zones, in order to provide flexibility to the cloud provider. Whether the enforcement of the policy is local or distributed, the policy must be applied in a consistent manner.

Finally, we don't specify the size of the possible zones inside the data center. First of all, because it is very difficult to obtain information on the size of existing data center, thus to build a coherent solution regarding the reality of its use. Furthermore, the core of our research resides in the enforcement of security policies, therefore we believe the search of scalability solutions is out of scope. Last but nor least, our solution will focus only on the feasibility and its performance will not be optimized.

3.3 Technical Alternatives

In life in general and in computer science in particular, there are several ways to achieve a goal. In this section, we review the different technical solutions available in order to enforce security policies by routing the traffic through security appliances.

For each requirement, we will determine the pros and cons of each solution. Then, we will explain our choices regarding the solutions which we will use in our project.

Isolation

We have identified the isolation of traffics as the first requirement of our model. First of all, we have seen the topological segmentation, where elements are physically separated

from one another. This technique provides a guaranteed isolation but impairs greatly the scalability of the solution as well as its cost. The use of VLAN is common in the networking field, as its use is predominant in enterprises and the research field. The principle is to create smaller broadcast domains inside a big network, in order to isolate their members from one another. Although this technology is useful in several fields, its lack of scalability and its manual configuration makes is little adapted for the cloud. The last technology evoked was the isolation and processing of the traffic by analyzing the characteristics of the flows. These techniques provide the isolation capabilities required and the management can be automated and centralized. The automation of the setting of the routing rules could also appear as a drawback, as the isolation is depending on the good programming of the controller, which could represent a single point of failure.

In order to build our solution, we believe the more suitable technology would be the last one exposed. Indeed, matching the flows directly would allow us to have a centralized control on the architecture, which is key to our problem.

Therefore, our architecture will be built as a SDN, and more precisely with the OF technology [11]. It will allow us to build a controller which will allow the automatic enforcement of rules. The OF technology is based on the separation of the control plane and the data plane. The centralized controller set rules in the forwarding elements, and the routing is done based on these rules. There is no superior logic inside the forwarding elements. This allows the configuration to be done in a centralized way. Furthermore, any packet belonging to an unknown flow will be sent to the controller, which will then set the rules inside the switches.

Security Chain

Our solution aims to secure the traffic of the tenants by making it traverse sequences of middleboxes, based on the tenant's requirements. In order to do so, we will need to route the packets from a source to a destination, and throughout the adequate number of middleboxes. Previously, we surveyed what have been done in the recent works in order to do so. In this section, we investigate the different ways to set the routing rules into the forwarding elements.

As the first packet of a flow is emitted by a VM, it reaches a forwarding element of the architecture. As it is belonging to an unknown flow, it is forwarded to the controller for processing. As we are in the context of a software-defined network, we have the possibility to set routing rules inside the forwarding element which originally sent the packet to the controller, but not only. Indeed, the controller can set rules in any switch it is connected to.

Thereby, we have two technical alternatives here, as we can set the rules either step-by-step, or hierarchically.

Enforcing the rules in a step-by-step manner would be likely to reduce the complexity of

the controller's logical. However, for the first packet of the flow, at each step of the route, the packet will be sent to the controller where it is going to be processed. In a hierarchical manner, several rules will be pushed down to multiple switches at the same time, and this, upon the first call to the controller.

Incidentally, the processing time of one packet could be higher in a hierarchical rule installation. However, the multiplication of requests of the step by step manner will likely be less efficient. Moreover, the communication between the controller and the forwarding elements will be more important in a step-by-step installation versus a hierarchical rule installation.

The way we enforce the rules is as important as the way we match the flows. Indeed, in order to set the rules, we need to determine what flows we need to process. The Openflow technology allows to match the flows based on many characteristics. It allows the switches to modify information on a packet, such as their addresses and tags.

Matching the flows could be done in two ways. On one hand, we could use the characteristics of the flow, such as the destination and/or source addresses. On the other hand, meta-data could be added to the packets, by using the OF capability of adding and removing tags.

The routing based on the characteristics is what is achieved in [19]. In this solution, they use the MAC address of the sender and receiver, in order to match the adequate security policy. The packet is then consecutively encapsulated with the MAC address of each middlebox along the path. At the end of the path, the packet is routed to its original destination.

Adding meta-data to the packet consists in adding a piece of information, using a tag, for example, in order to identify the security policy which applies to the flow. The advantage of this solution lies in the controller. Indeed, having many source/destination combinations matching the same middlebox sequence would require a lot of memory in the controller. Furthermore, retrieving this data would increase processing time.

In order to build our solution, we first choose to use the hierarchical rule installation, as we believe it is a simpler approach and will reduce the communications between the server and the forwarding elements. Furthermore, we believe it will reduce the overall processing time of the controller as well. Moreover, the routing will be done based on the meta-data. Indeed, we believe that it is more efficient to preset several middlebox sequences and to bind a specific identifier to each. Henceforth, we will call this identifier Application ID (AppID).

We assume that the running hypervisor has the capability of inserting this AppID to the packet, based on the application running on the VM.

As the VM emits a packet, the first encountered switch will recognize this packet as a new flow, forward it to the controller. The latter will then extract the AppID in order to

retrieve the corresponding security policy in the form of a middlebox sequence. Immediately, the controller will push the routing rules down the forwarding elements along the path from the source to the destination, and throughout the middlebox instances.

The routing will then be made by adding meta-data to the packet in order to route it to the next middlebox instance. This data, most probably in the form of a tag, will represent the middlebox instance to be routed to. At the switch level, the matching of a flow will be done based on this tag. It is thus necessary to keep a table binding the middleboxes instances to their meta-data tag. We decide to call the meta-data tag the Elastic Enforcement Layer (EEL)-tag.

Migration Tolerance

Once a security chain is in place, it needs to stay coherent. This is a challenge in a context where the nodes can migrate from one physical server to another. This is why the requirement we tackle in this section is the migration tolerance.

Thereby, we want to build our solution having in mind that anytime and anywhere in the network, one node could disappear, and suddenly, reappear in another part of the network. It means that the rules which have been set prior to the migration of a node should not match other packets from other nodes, whether they were already running or suddenly appearing. It also means that once the VM has migrated successfully, its newly sent packets must be routed through the same middleboxes as it was prior to the.

Regarding the previous point, the first approach would be to discard the rules which match the information of the migrating node, as its MAC or IP address, for instance. The rules, instead of being discarded, could also be modified. Another method would be to use the default timeout of the matching rules inside the flow tables of the OF-switch and most importantly, to use a way of configuration/reconfiguration allowing the new flows to be routed efficiently. It would be only a matter of time for the obsolete rules to be discarded automatically.

The instant deletion of obsolete rules is quite interesting as it allows to be sure that all the rules in the data center are coherent and up to date. However, it requires firstly to be able to detect the migration. This requires for the cloud operating system and the controller to be able to communicate somehow. Otherwise, the detection could be made by analyzing the network topology changes. It would mean that for a node which just start emitting packets, the controller may consider this as a change of topology (as the node appears in the network) and unicast each forwarding elements of the network in order to discard potential rules matching this newly emitting node. This is not a good solution as it would flood the network and thus decrease the scalability of the architecture. In order to avoid this, we could

keep a mapping between the nodes and the switches in which rules corresponding to the nodes are set. However, this model is not satisfying either, as maintaining such a table would be wasted memory and also would be a hurdle to an optimal scalability. Finally, another drawback of this solution is the fact that a receiver node would not necessarily emits packets, and thus would not be detected as a migrated node.

We believe the best solution is to find a way to configure the route in such manner that packets will be routed accordingly to the security policies, and this, whether it's a VM newly created of freshly migrated. Upon migration, the obsolete rules will be discarded as they reach the set timeout. As the migration of a receiver node would imply a communication with the cloud operating system, we decide to focus solely on the migration of a sender.

Scalability

Scalability is a key stake in our project, as it is a crucial issue for all cloud architectures. We do not set numbered requirements, neither regarding the optimal size of our network, nor the maximum number of rules we want to have in our switches. We only set a first stone in order to build an architecture allowing the dynamic enforcement of security policies in a multi-tenant cloud network. However, in order to design a solution which could be enhanced in the future, we need to take the scalability of our solution into consideration. More precisely, we consider a cloud network which can be partitioned in zones, each of them being ruled by a distinct controller. Furthermore, we want to set rules which match more than one source-destination combination, in order to put less burden on the forwarding elements and thereby reduce their processing time.

As we have exposed in the section 3.1.3, it is possible that in such a context, we need to apply a security policy between two nodes belonging in different zones of the data center. Thus, we want the rules to be set in a way which allow to apply one part of the policy in one zone and the rest of it in another zone. The matching of the flows is going to be based on meta-data added to the packet by the forwarding elements, the EEL-tag. The presence of an EEL-tag on the packet while it reaches the second zone could be used as an information meaning that the security policy has already been partially applied. A security policy, as we previously said, is a chain of middlebox type to be traversed. After careful reflexion, we believe the EEL-tag can be divided in two sub-categories. The first category will represent the middlebox type. A particular tag would mean "firewall", "DPI", "Intrusion Detection System (IDS)", etc... A security policy will then be declared as a chain of EEL-tag representing a middlebox type. We will call these tags : generic EEL-tag (gTag). The other category of EEL-tag would represent the middlebox instance, and be used as the routing rules are pushed down the forwarding elements. We will call these tags : instance EEL-tag (iTag).

As the packet must be routed from one zone to another, and in the case where just a part of the security chain has been applied, the gTag of the next middlebox type to be traversed will be set to the packet. As the packet arrives in the second zone, the controller will gather the gTag information in combination with the AppID, which will give the security chain as well as the part of it which has previously been applied.

The use of these EEL-tags will allow us to further obtain more scalability. Indeed, considering the source VM1, to destination VM2 and going throughout M1, M2 and M3. Once the tags have been set, the routing is made based on the iTags corresponding to M1, M2 and M3. These iTags will match the packets from VM1 to VM2, but also any other source-destination combination which would be routed to M1 and/or M2 and/or M3. This allow a flow aggregation, unlike, for example, a routing based on the source and destination IP or MAC addresses.

The use of these tags will allow the partition of the network in smaller zones, each ruled by a separate controller. The routing is done from a machine to another, throughout the corresponding middleboxes, independently to the location of the machines and the middleboxes. Furthermore, the use of these tags allow us to reduce the number of entries in the flow tables of the switches, as all packets going to a particular middlebox would be treated the same way.

Conclusion

In this section, we built on the refined requirements, in order to achieve the transition from the definition of the project's requirements to the technical solutions we are opting for.

We have exposed the reasoning underlying the choices we have made regarding all the requirements.

3.4 Selected the technologies to build our algorithm

Our goal is to automatically enforce security policies in a multi-tenant cloud network. The environment is volatile as the nodes can migrate unpredictably. The environment is elastic as the network can scale up and down to answer the customers' needs.

3.4.1 Selected Technologies

In this section, we summarize the selected technologies and models that we will use in order to build our solution.

We build a solution based on the OF technology. A centralized controller will operate on a set of switches which will send to the controller any new packet belonging to a new flow.

As the controller receives this packet, it retrieves the AppID inserted in the packet by the hypervisor. This AppID corresponds to a security policy, in the form of a chain of middlebox type. This chain of middleboxes is represented by a chain of gTags, also called gTags. These gTags are then translated in iTags, each iTag corresponding to an actual running middlebox instance of the network. The controller then identifies the path from the source to the destination and throughout the selected middlebox instances. The routing rules are pushed down in all the forwarding elements along the path. The iTags are pushed by the switches to the packet, in order for them to be routed to the adequate middleboxes. The use of those tags allow us to aggregate many flows with one rule.

In the case where the destination and source VMs are in different zones, ruled by different controllers, it can happen that part of the chain is applied in the first data-center. Then, the gTag corresponding to the next middlebox type to be applied is added to the packet, and the latter is routed out of the zone. As the packet reaches the second zone, the packet is sent to the controller. The latter analyzes the information present in the packet and finds the AppID as well as the gTag. The controller can then deduce the rest of the security chain and set the routing rules down the data path. In order to avoid ambiguous resolving of security policies left to be traversed, we need :
$\forall(gTag_i, gTag_j)$ in the same chain of middlebox, $gTag_i \neq gTag_j$.

3.4.2 Algorithm Flowchart

We now have a clear idea of what we want our solution to do. However, in order to have a better view of it, we want to build the flowchart of the algorithm. This flowchart will help us visualize our solution in order to efficiently build a proof of concept of our network controller. The flowchart presented on 3.5 summarize the logic of our solution. Below is the detailed explanation of the algorithm

The initial situation is a VM running on any physical server. This VM runs an application which corresponds to an AppID. As the VM starts emitting packets corresponding to this application, the hypervisor intercepts the packets and set an AppID to the packets. We choose not to precise how this is achieved as we consider the intervention of the hypervisor being out of scope. The packet then reaches a first switch, where it is recognize as belonging to none of the existing flows, because it does not match any flow entry inside the OF switch. The controller, having the capability of analyzing the packet and deciding what flow entries to push down the data path, receives this unknown packet. It then extracts the AppID tag contained in the packet. The controller holds a table matching AppIDs to security chains in form of gTag chains. The AppID retrieved from the packet will then reveal the security policy to be applied for this particular flow. We have seen previously that in the case where the

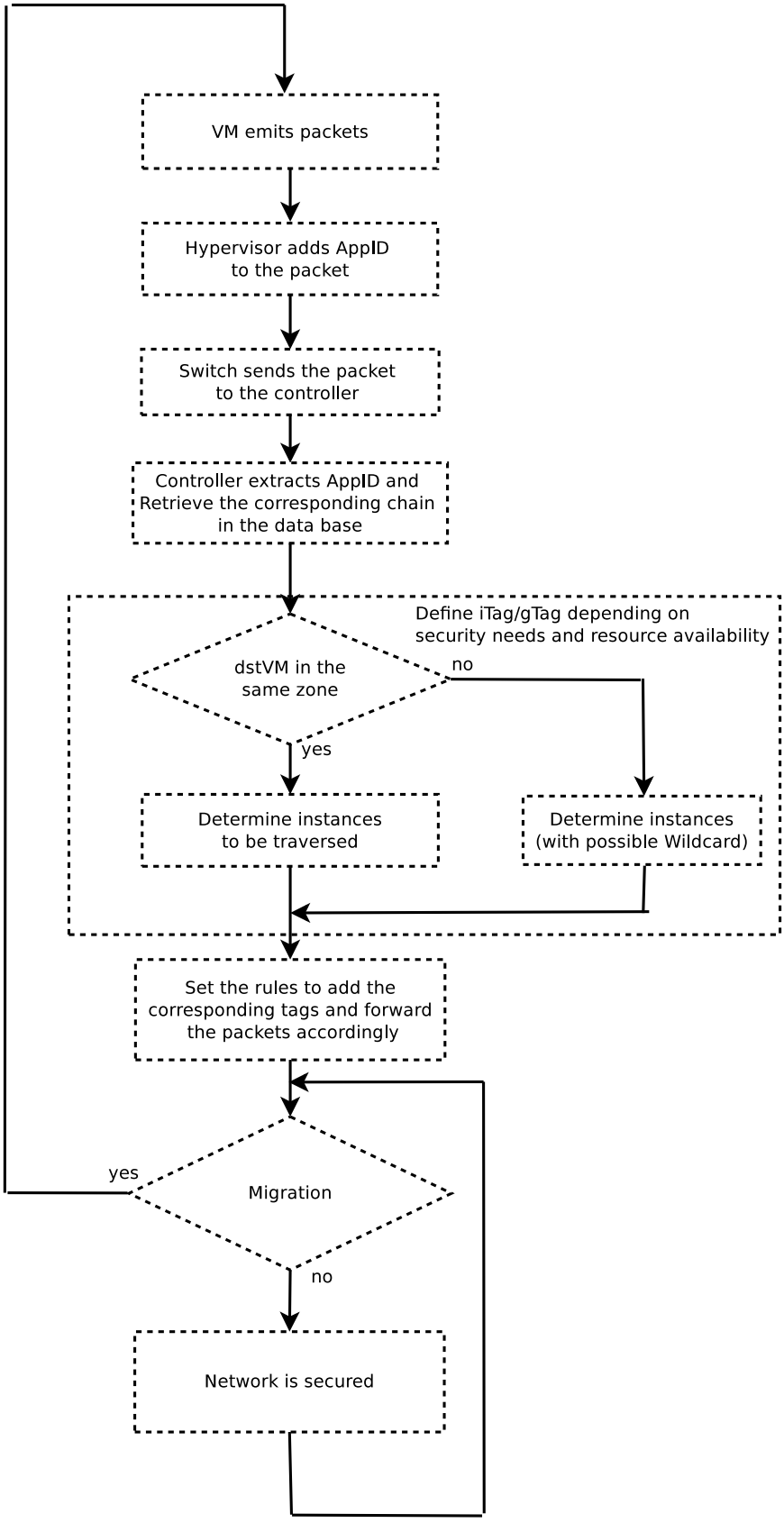

Figure 3.5 EEL algorithm flowchart

destination and source VMs are located in different data centers, we have the possibility to delegate the application of all or part of the security policy to the controller of the destination zone. This particular situation will be referred to as a "wildcard". This wildcard possibility is obviously impacting the application of the security chain, it is why the next step in our algorithm is to check whether the destination VM is located in the zone ruled by the very same controller which received the packet belonging to an unknown flow. The location of the destination machine will impact the conversion of the gTag chain to a iTag chain. Indeed, when the destination is remote, it is possible to use a wildcard. when a wildcard is used, the gTag of the first middlebox remaining to traversed will be used in order to tag the packet before routing it out of the zone. Once that all the parameters have been determined, the controller pushes the rules down all the switches along the path. Once this step has been completed, the security policy is enforced. Finally, upon migration, the same process is repeated which allows the security policy to be applied even though the security instances to be traversed may differ from the ones which were traversed prior to the migration.

3.4.3 Revisited Use Cases

In this section, we revisit the use cases previously exposed by applying the logic described in the flowchart.

A use case of security rule enforcement

To illustrate how our model works, we describe in this section a simple use case to illustrate how we can enforce the traversal of a sequence of middle boxes between a pair of VMs. The example is demonstrated in Figure 3.6. Let's assume that the tenant has defined that the traffic emitted by the VM1 correspond to the AppID-9, which means that the traffic must go through an IDS, an Application Firewall (AppFW) and a DPI. Let's assume that VM1 is trying to communicate with VM2, which is in a region controlled by the same controller. Note that in our examples, the ingress and egress switch of middleboxes are the same.

1. VM1 starts emitting packets. These packets are intercepted by the hypervisor that inserts the AppID into the IP options field.
2. The OpenFlow-Switch (OFS) forwards the first packet to the controller.
3. The OpenFlow-Controller (OFC) extracts the AppID and determine the chain of gTags to be traversed
4. It then matches the Generic Tags (gTags) to an Instance Tags (iTags) range

5. It then chooses the middlebox instances to send the packet to (based on cloud resource availability). In our example, let's assume the chosen instances of IDS, AppFW and DPI correspond to iTags 2070, 1045 and 3093 respectively.
6. The OFC adds a two new flow-entries into the VM1's edge OFS :
   - Packets from VM1 (to VM2) must be tagged with EEL-tag 2070.
   - Packets with EEL-tag 2070 must be routed to the next switch towards the IDS 2070 instance.
7. The OFC also adds three new flow-entries into the IDS's ingress and egress OFS :
   - Packets tagged with EEL-tag 2070 must have their tag popped and be forwarded to the IDS (ingress).
   - Packets out of the IDS, from VM1 and to VM2 must have the EEL-tag 1045 pushed (egress).
   - Packets with EEL-tag 1045 must be routed to the next switch towards the AppFW 1045 instance (egress).
8. Similar rules to the previous ones are to be set into all the middleboxes edge's OFS. Note that for the egress switch of the last middlebox in the chain, the packet should only be routed to the next switch towards the destination VM.
9. Along the path, the controller adds a rule to forward the packet to the next switch towards the middlebox instance, based on the EEL-tag.

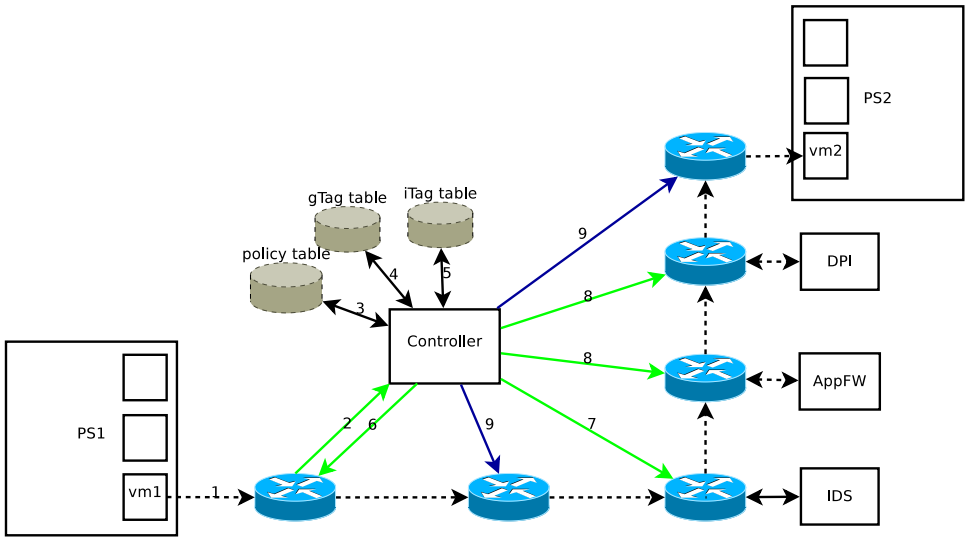

Figure 3.6 Simple use case of policy enforcement

In this simple use case, we have illustrated how a simple chain of middleboxes is set for a particular flow.

A Use Case of Aggregation

Next, we demonstrate the scalable design of our proposal under aggregation. This example is a variation of the previous one. We have now three source VMs communicating with the same destination VM. It is shown in Figure 3.7. Let's assume that the tenant has defined that the traffic emitted by the VM1 and VM2 correspond to the AppID-9, which means that the traffic must go through an IDS, an AppFW and a DPI and traffic emitted by the VM3 correspond to the AppID-10, which means that the traffic must only go through an AppFW and a DPI. Let's assume that these VMs are trying to communicate with VM4, which is in a region controlled by the same controller.

1. The VMs start emitting packets. These packets are intercepted by the hypervisor that inserts the AppID into the IP options field.
2. Same as previous.
3. Same as previous.
4. Same as previous.
5. Same as previous.
6. The OFC adds new flow-entries into the VM1's edge OFS :
   - Packets from VM1 (to VM4) must be tagged with EEL-tag 2070.
   - Packets from VM2 (to VM4) must be tagged with EEL-tag 2070.
   - Packets from VM3 (to VM4) must be tagged with EEL-tag 2070.
   - Packets with EEL-tag 2070 must be routed to the next switch towards the IDS 2070 instance.
7. The OFC then adds new flow-entries into the IDS's ingress and egress OFS :
   - Packets tagged with EEL-tag 2070 must have their tag popped and be forwarded to the IDS (ingress).
   - Packets out of the IDS, from VM1 and to VM4 must have the EEL-tag 1045 pushed (egress).
   - Packets out of the IDS, from VM2 and to VM4 must have the EEL-tag 1045 pushed (egress).
   - Packets with EEL-tag 1045 must be routed to the next switch towards the AppFW 1045 instance (egress).
8. Similar rules to the previous ones are to be set into all the middleboxes edge's switch. Note that for the egress switch of the last middlebox in the chain, the packet should only be routed to the next switch towards the destination VM.
9. Along the path, the controller adds a rule to forward the packet to the next switch towards the middlebox instance, based on the EEL-tag.

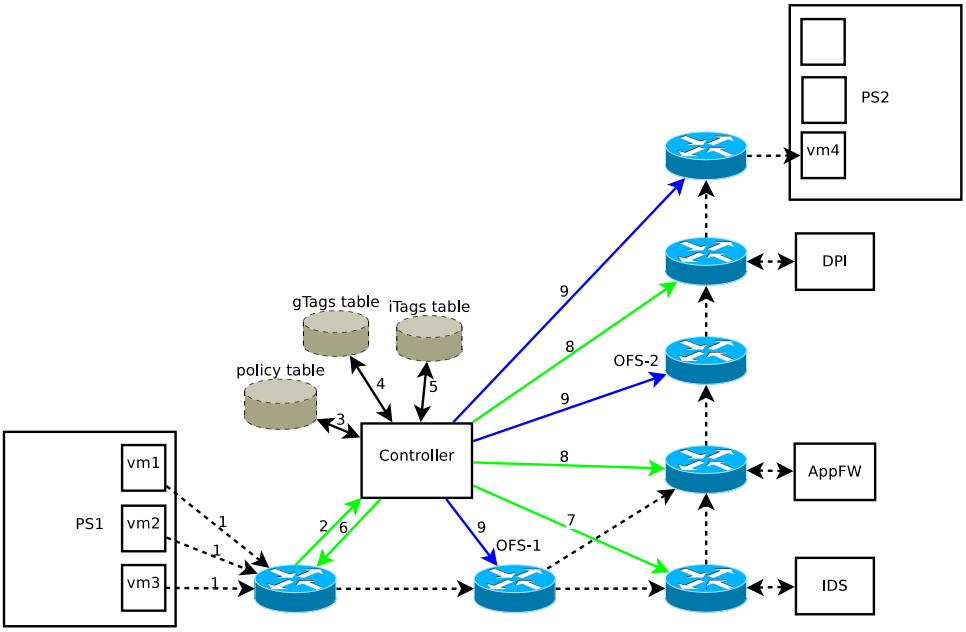

Figure 3.7 Simple use case of flow aggregation

In this second simple use case, we illustrated how several flows can be aggregated. Particularly, the rules at a middleboxes ingress switch allow to match all the flows directed to this particular middlebox, in a simple manner. In the ingress switches, only one rule has to be set, matching the EEL-tag. Furthermore, we can see that the routing based on the EEL-tags allows to aggregate several flows on the switches along the path. In our example, only two rules are pushed into OFS-1 (Openflow Switch-1) and only one rule at OFS-2, for a number of three different flows.

A Use Case of Wildcard Tags

Our design also makes use of the wildcard in rule matching to improve the scalability. The detailed steps are shown in Figure 3.8. It is similar to previous example. However, in this case, the source and destination VMs are in zones ruled by different controller.
The chain of middleboxes remains IDS, AppFW a DPI.
The gTag corresponding to an AppFW is 18.

1. VM1 starts emitting packets. These packets are intercepted by the hypervisor that inserts the AppID into the IP options field.
2. The switch forwards the first packet to the controller.
3. The OFC extracts the AppID and determine the chain of gTags to be traversed
4. It then matches the gTags to an iTags range

5. The controller chooses IDS instance 2070, but based on the availability of AppFW-1, the controller-1 decides to use a wildcard tag for the AppFW, and forwards the packet towards its final destination.
6. The controller adds a two new flow-entries into the VM1's edge switch :
   - Packets from VM1 (to VM2) must be tagged with EEL-tag 2070.
   - Packets with EEL-tag 2070 must be routed to the next switch towards the IDS 2070 instance.
7. The controller also adds three new flow-entries into the IDS's ingress and egress switch :
   - Packets tagged with EEL-tag 2070 must have their tag popped and be forwarded to the IDS (ingress).
   - Packets out of the IDS, from VM1 and to VM2 must have the EEL-tag 18 pushed (egress).
   - Packets with EEL-tag 18 must be routed to the next switch towards VM2 (egress).
8. Along the path, the controller adds a rule to forward the packet to the next switch towards the middlebox instance, based on the EEL-tag.
9. Once the packet reaches the first switch of the second region, the packet is sent to the controller-2.
10. It resolves the AppID in order to know what types of middlebox are to be traversed.
11. It then resolves the gTags in order to know what types of middlebox are still to be traversed.
12. The controller chooses the middlebox instances to be traversed.
13. Rules are set same as previous.

In this third simple use case, we illustrated the use of wildcards. We can see that a controller has the ability to delegate the responsibility of applying the security measure to another region. This can be very useful in the cloud, as it allows a better use of resources.

A Use Case of Migration

Our design also provides the resilience of security measures upon VM-migration. The detailed steps are shown in Figure 3.9. It is similar to first example. However, the chain of middleboxes is now IDS and AppFW.
The iTag corresponding to the IDS-1 is 2070. The iTag corresponding to the AppFW is 1045. The iTag corresponding to the IDS-2 is 2080.

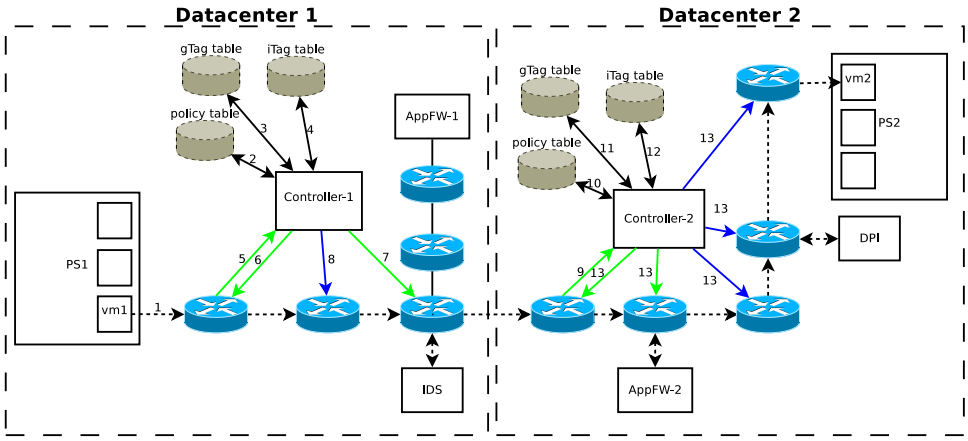

Figure 3.8 Use case of wildcards

1. VM1 starts emitting packets. These packets are intercepted by the hypervisor that inserts the AppID into the IP options field.
2. The switch forwards the first packet to the controller.
3. The OFC adds a two new flow-entries into the VM1's edge OFS :
   - Packets from VM1 (to VM2) must be tagged with EEL-tag 2070.
   - Packets with EEL-tag 2070 must be routed to the next switch towards the IDS 2070 instance.
4. The controller also adds three new flow-entries into the IDS's ingress and egress switch :
   - Packets tagged with EEL-tag 2070 must have their tag popped and be forwarded to the IDS (ingress).
   - Packets out of the IDS, from VM1 and to VM2 must have the EEL-tag 1045 pushed (egress).
   - Packets with EEL-tag 1045 must be routed to the next switch towards the AppFW 1045 instance (egress).

   A similar rule is set on the AppFW's ingress and egress switch
5. Along the path, the controller adds a rule to forward the packet to the next switch towards the middlebox instance, based on the EEL-tag.
6. The VM1 migrates from the physical server PS1 to the physical server PS3 and becomes VM1'.
7. VM1' starts emitting packets. These packets are intercepted by the hypervisor that inserts the AppID into the IP options field.
8. Same as previous.

9. Same as previous. Note that the IDS iTag is now 2080. Only the AppFW egress switch rules may be modified, for example if VM1 and VM1' don't have the same MAC address.
10. Same as previous.

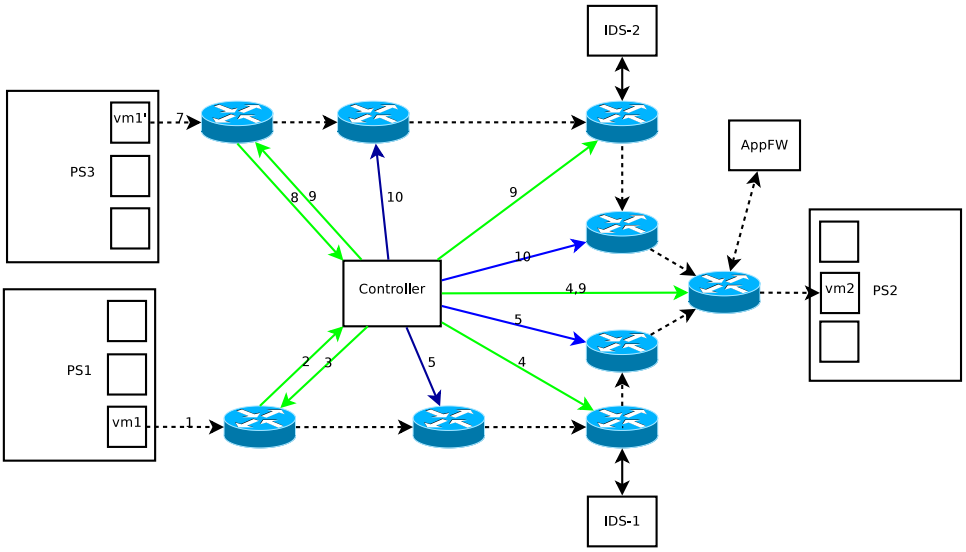

Figure 3.9 Resilience upon VM-migration

In this last simple use case, we illustrated how the security measures persist through migration. New rules are enforced when the VM1' starts emitting packets, in the same way as previous. The former rules matching the VM1 as the destination will timeout in order to reduce the load on the switches. Even though the security appliances traversed are not the same as they were prior to the migration, we can see that the security chain is respected.

CHAPTER 4

IMPLEMENTATION

There is a time to read and a time to write, there is a time to think and a time to act. We first described our project and reviewed the literature on the relevant work that had been done during the past few years on the matter. We then refined our requirements and proposed a new model to control the network. This model is designed in order to enforce security policies in the form of middlebox sequences. The enforcement of these policies is constrained by the characteristics of the cloud and particularly the migration of nodes. It is now time to develop an implementation of our algorithm, because the theory is not enough. In order to create a prototype, we need to make further choices. The current chapter will describe our implementation. We will explain what we want to achieve by building this implementation as well as the proof we want to find. We will then explain what the environment we are working in. We will close this section by describing the created components and the choice we have made in order to go from theory to practice.

4.1 Implementation

When we transpose a model or an algorithm into a concrete solution, it is always a challenge. Indeed, on one hand we want to stay close to the algorithm but on the other hand, we must make decisions regarding the implementation but also regarding the developing tools to use. After designing the architecture allowing a cloud provider to secure its tenants networks, it is now time to implement and test it. In order to do so, we first define what information we want to gather from the tests. We will then focus on the environment in which we will build our solution. Finally, we will describe the created components of our prototype.

4.1.1 Goal of the Implementation

The work that we try to achieve is something that has not been done in the past. One work is similar to what we try to do, as it applies the traversal to a sequence of middleboxes to certain traffics. What we want to achieve is however different, as we focus on the consistency of the rules upon node migration. Some other work has been done in order to secure the cloud, but the focus was more on the ease of management or the performance of the network controller. As this is the first implementation of our design, we will focus more on feasibility

and functionality, rather than optimized performance. We can further argue that our design is dependent on routing protocol in order to work, as our work focuses only on the mechanism allowing the migration tolerance. The routing protocol that we will use will not be optimized. Furthermore, the rule optimization is another point which will not be addressed in our research. We will thus build an implementation and test it in order to provide the proof that our mechanism can achieve what we built it for. We want to build simple cases as we believe they will be present our model in a simpler and more general way.

Secure the Traffic by Enforcing the Traversal of Middlebox Sequences

The proof of concept will first have to demonstrate our first requirement. We will show that our implementation is able to enforce the traversal of a middlebox sequence. It means that based on the Application ID inserted in the packet by the hypervisor, the network controller will retrieve the security chain corresponding to this particular flow. The flow must then be routed throughout the middleboxes, respectively to the order of the security policy. Moreover, the packet must not be encapsulated or modified as it reaches the middlebox, in order for the packet analysis to be done correctly. Once that the whole security policy has been applied, the packet must be routed towards its original destination. As the packet reaches the first switch and is sent to the controller, the latter must set the rules along the whole path. It means that for one flow, the communication between a switch and a controller must be done only once, when the first packet reaches the first switch.

Resisting Node Migration

The second point that we need to demonstrate that our solution can operate in the cloud, despite the node migration. In order to do so, we will need to migrate the nodes during our tests. We want the node migration to require the configuration of a new route. The only requirement is that the new route has to respect the security chain. Indeed, as the freshly migrated machine emits a packet, the same process will occur at the hypervisor level in order to apply the AppID tag. Then, the route will be configured throughout the proper middleboxes as previous. Regarding the security instances, the whole route, or just a part of it, could be different from the previous route. We plan on providing such a proof in our test.

Scalability Mechanisms

We do not tackle the scalability issue of our model. Nonetheless, our model takes in consideration the partition of the data center in smaller zones. Each zone being ruled by a specific controller. There is a possibility that two VMs which are communicating are

physically located in different zones. We have studied that case in the section 3.1.3 and we have designed a solution allowing the traversal of the middlebox chain to be delegated to another zone. We want to provide the proof that when such a situation occurs, the delegation of the entire policy of part of it can be done from one zone to another.

4.1.2 Tools

This section is describing the design and implementation of our prototype. In order to do so, we first need to set up the environment in which we are going to work. We also need to select the tools that we are going to use.

The deployment of the solution as well as the tests will be realized in a virtual environment, in order to have a greater liberty as we would have had on a physical host. It allows us to develop the project on several physical machines concurrently, as we only need to adjust the modifications on the VM.

The two leading virtualization software are VirtualBox, developed by Oracle, and VMWare Workstation, developed by VMWare. We chose the first one because it is a solution which is under and open license whilst providing the same capabilities of its concurrent, regarding our requirements.

The Virtual Machine that we will use for development is under the Ubuntu operating system. It has 512Mo of flash memory and a 6Gio hard drive.

Our solution is based on Software-Defined Networking. As it comes to the selection of which SDN protocol to use, the choice is not too hard to make. Indeed, the OF protocol is the leader in this field, as it is the most advanced and complete protocol. It is standardized by the Open Network Foundation and is already implemented by many vendors like Cisco, IBM, Juniper, HP and NEC.

Now that we know what protocol will allow us to control and rule the network, we need to choose which controller we are going to use in order to do so. There are several existing controllers in order to control OF switches. Their differences are mostly based on the programming language they offer.

The first controller choice is NOX. NOX is a controller platform on top of which we can write a controller in Python, C++, or some combination of the two. NOX was the first OF controller and was developed by Nicira Networks. It was then been donated to the research community in 2008 and has been the basis for many and various research projects.

Beacon is a Java-based OF controller platform. The development of a controller is encouraged to be done using the Eclipse Integrated Development Environment, and best run in a host environment rather than a VM.

Floodlight is another Java-based OF controller platform. It is Apache-licensed and claims

to be an enterprise-class OpenFlow controller.

Lastly, the Trema OpenFlow controller is a framework for developing controllers in Ruby and C. Trema is the youngest of the controllers but provides a testing environment in the form of a network emulator.

In order to develop our solution, we chose the NOX controller platform. We made this choice because it was the best documented platform on which the community was very active. We chose this controller because it seemed to be the one who fitted the best our needs at the time, but the project could have been implemented on any other platform.

By the time our project ended, the NOX controller platform has been divided in two distinct projects. NOX, on top of which the controller can be written in C++ and POX, on top of which the controller can be written in Python. The previous NOX version is referred to as NOX-classic.

4.1.3 Created Components

Our project consists in the creation of a intelligent network which allows the enforcement of security policies. Thanks to the recent development in Software-Defined Networks, the implementation of such a project can be done in a much more easy way than it used to be.

Indeed, the mechanism that we developed in order to provide the consistency of the security policies in spite of the migration of the nodes can be implemented as a network controller.

In this section, we explained the choices we made as we coded the solution. The flowchart of our algorithm can be found on the 3.5. The first steps are not the responsibility of the controller.

As the packets arrives to the controller, the latter needs to retrieve the middlebox sequence. In order to do so, we need to get the AppID from the packet and to match it against the databases containing this information.

Thereafter, we need to evaluate whether or not the source and destination VMs are in the same zones. If it is not the case, measures will have to be taken in order to modify the security chain accordingly.

As we know the source of the packet, its destination and the middleboxes throughout which it must be routed, we need to calculate the adequate route.

Finally, we need to build the rules in order to treat the packets adequately all along the path and to enforce the routing rules. After doing all these steps, the enforcement of the security policy will be complete.

Retrieving the Security Chain

As an unknown packet is detected at the switch level, it is sent to the controller. The first thing that the controller must do is to extract the AppID in order to retrieve the security chain.

We assume that the hypervisor has the capability to insert the AppID in the packet, but we don't specify a precise location for this identifier. However, in order to implement our solution, we have to choose a location. We decide to consider this Application ID inserted in the IP options field of the IP header, as shown in the 4.1.

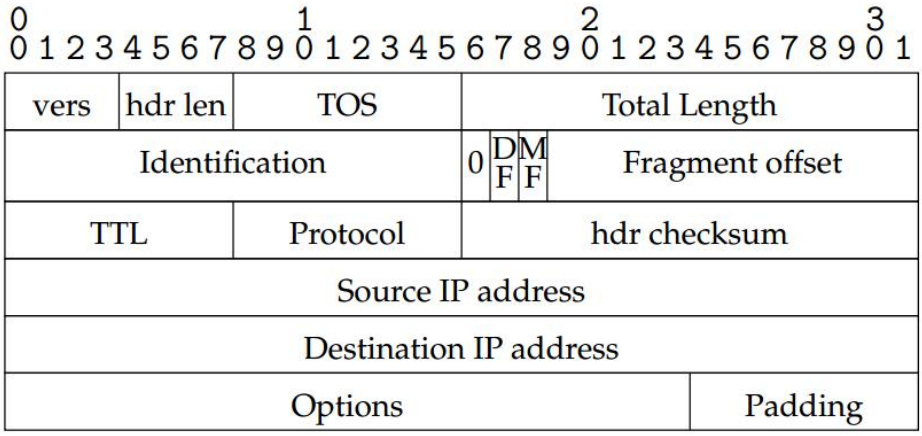

Figure 4.1 IP header

At the controller level, we need to dig in the packet in order to retrieve this ID. The python library does not offer a library able to get or set this field. In order to achieve it, we use the libraries *array*, *struct* and *socket*. We parse the packet in order to access the different fields of the Ethernet and IP headers, in order to fetch the value of the IP options field.

When we have the AppId, we need to translate it into a security chain. As we explained, we use gTags in order to designate a middlebox type. Thereby, a security chain is represented by a list of gTag. This is the first database kept in the controller, a dictionary where each AppID (the key) corresponds to a security chain in the form of a gTag list.

The i^{th} line of the table is as follow :

$$AppID_i \rightarrow (gTag_{i,1}, ..., gTag_{i,j}, ...gTag_{i,n})$$

From the security chain to the particular instances to be traversed, the road is not over yet. We know that an iTag represent an instance of middlebox. In order to know what type of middlebox this iTag corresponds, we decided that to each gTag will correspond an iTag

range. For instance, considering a particular gTag designating a firewall, all iTags included in the range corresponding to a gTag will be firewall instances.

The second database kept in the controller is thus a dictionary mapping gTags (the key) to an iTag range.

The i^{th} line of the table is as follow :

$$gTag_i \rightarrow (iTag_m, iTag_p)$$

The road is almost complete towards the resolve of the middlebox instance. We have the range of possible iTags corresponding to the particular middlebox we want to traverse, but it needs to be linked to an actual running Virtual Machine somewhere in the topology. Therefore, our third database will be a dictionary mapping the iTags (the key) of the running middleboxes to the IP address of the running machine.

The i^{th} line of the table is as follow :

$$iTag_i \rightarrow ipaddress_i$$

In this section, we explained how we parse the incoming packet in order to fetch the AppID and how we further translate it from an AppID to a chain of actual instances to be traversed.

Wildcard Possibilities

As we keep progressing in the flowchart presented in page 3.5, we now go one case further. We now need to determine whether or not the source and destination machine belong to the zone ruled by the controller.

In order to do so, we must be able to know the topology of the network, in real time. There is a built-in topology module in NOX but it does not allow us to use it in a way that is useful to us. We decide to build a table where we store the IP addresses of the Virtual Machines bound to the ID of the switch they are connected to as well as its port. Indeed, each switch is assigned a Datapath-ID (dpid).

Let A_i be the IP address stored at the i^{th} line.

$$A_i \rightarrow (dpid_i, port_i)$$

This way, we will know right away if the source of the destination machine belongs or not to the ruled zone.

In order to figure out where to send the other packets to, we need to know where are the gateways to these other zones. In order to keep track of these gateways, we implement

a register in a same way it's done in classic networking. We set a default gateway and we build a dictionary where the key is the IP address of the foreign machine, bound to the corresponding dpid.

Let B_i be the IP address stored at the i^{th} line.

$$B_i \rightarrow (dpid_i, port_i)$$

These mechanisms allow us to know whether or not the source and destination machines are located in the zone ruled by the controller.

In case the destination machine does not belong to the current zone, the dpid and the port of the gateway switch will be stored and a Boolean value representing the wildcard possibility will be set to true.

In case the source machine does not belong to the current zone, the controller must check if the packet already has an EEL-tag. If that's the case, this EEL-tag must be extracted. This tag is a gTag and thus represent a type of middlebox. Once it is retrieved, the security chain will be trimmed and only the elements of the security chain which haven't been applied yet will remain. For example, a security chain is composed of three gTags : *(gTag1, gTag2, gTag3)*. The packet arrives tagged with the second gTag, gTag2, the controller will remove the first gTag and the new security chain will be *(gTag2, gTag3)*.

The EEL-tags must be inserted to the packet by the switches. The OpenFlow protocol 1.0 only allows the modification, pushing and popping of the VLAN tag. With 4096 values, we can use the first hundred for the middlebox type (i.e. the gTag), and the rest of the values for the middlebox instance (i.e. iTag).

Therefore, the EEL-tag is retrieved by the controller by checking the presence of a VLAN tag on the packet.

Determine the Route

Once we figured out the security chain and whether or not the destination machine belongs to another zone, we must determine the route that the packet must take in order to go to destination.

We chose to implement a very simple routing protocol. The source switch is known, as well as the switches connected to the middleboxes to be traversed, and the destination switch. We then subdivide the route in smaller segments and we calculate the shortest path between the different elements of the route.

In order to do so, we use the built-in topology discovery module. This module builds and maintain an adjacency list of the network links and periodically iterates over the discovered links of the network. This adjacency list is simply a table binding two switch and port information.

The i^{th} line of the table is as follow :

$$(dpid_{i,1}, port_{i,1}) \rightarrow (dpid_{i,2}, port_{i,2})$$

We use this table in order to build the shortest path.

However, we have seen that one gTag correspond to several iTags. Thereby, there are several possible path to go from one point to another.

Furthermore, in the case of a wildcard, the whole security chain does not have to be traversed, as we can delegate the application of a portion of the chain to the next zone. Therefore, if a security chain is *(gTag1, gTag2, gTag3)* and it's possible to delegate part of the chain to the next zone, then the possible middlebox chains to be traversed in the current zones are : *(gTag1, gTag2, gTag3), (gTag1, gTag2), (gTag1)* and even *()*. The last combination is the case where the whole chain is delegated to the next zone.

Moreover, for each gTag, there may be several active middlebox instances, represented by their iTag. In order to determine the route we must apply, we determine all the possibilities and we calculate the number of hops. The route with the lowest number of hops is chosen. In case of a wildcard, and if several combinations have the same number of hops, we choose the one with the highest number of middlebox traversed.

Create and Push the Rules

The last box of the flowchart if the most important one. It takes all the previous work and turn it into concrete routing rules in the data center.

In order to push the flow-entries down the switches, we use the functions defined in the OF library in nox, by calling *import nox.lib.openflow as openflow* in our controller.

In order to define a flow entry, we must define the matching attributes and the actions to be taken.

There are several types of rules that can be set. First, the rules matching the source and destination field, as well as the incoming port, when the first switch receive the freshly emitted packet. Let's assume that the middlebox chain consist in three tags : *(iTag1, iTag2, iTag3)*. The action taken by the first switch will be then to push the *iTag1* to the packet and to route it according to the path which has been calculated in the previous section.

The switches between the source switch and the first ingress switch will all have the same type of rule. Based on the *iTag1*, the packet will be routed to a particular port. This port has been calculated during the route calculation.

At the ingress switch of a middlebox, the rule will consist in matching one attribute and applying two actions. The matching attribute is obviously the tag corresponding to the middlebox. The packet must then be sent to the middlebox, but before that, the tag must be popped, so the middlebox receives an unmodified packet.

The egress switch of the middlebox will receive an untagged packet. Based on the source and destination field, as well as the port from which the packet arrived, the controller will have to push the $iTag2$ down to the packet. A second rule consist in matching this tag and to route the packet to the adequate port.

The rules set in the forthcoming switches are the same as previous, until we reach the last middlebox.

From the egress switch of the last middlebox to the destination switch, there are two cases. First, the destination machine is located in the current zone, and the destination switch is connected to the destination machine. In that case, the routing in the last segment is done by matching the destination address.

However, the destination can be located in another zone, in the wildcard case. In this situation, the tag of the next middlebox to be traversed will be added to the packet. The routing will then be made based on the tag and the destination address. If all middleboxes have been traversed, the routing will be done solely based on the destination address

Summary

We explained in this chapter how we implemented the prototype of our solution. The complete code of the controller can be found in the annex A.

Our prototype is not designed to provide optimized performances but only in order to provide a proof of concept. Getting this proof of concept is what we aim to do in the next section.

CHAPTER 5

TESTING AND EXPERIMENTAL RESULTS

The current chapter will cover the tests we want our implementation to go through. We will explain the scenarios that we created in order to gather the proof we want. We will describe the test environment as well as the tools we will use. We will end the section by describing the tools that we created in order to realize the tests.

After the implementation is done and the tests are created, it was time to run the tests on our implementation. For each point that we want to demonstrate, we ran the test, made observations on the behavior of the implementation. Thereafter, we analyzed the obtained results.

Once the analysis is done, it was time to adopt a larger view on our tests, our implementation and our project. We will expose their limitations as well as the improvements which can be brought to our model.

For each point that we treat, we want to provide a clear view on the choices we made, as well as the reasons that brought us to these choices.

5.1 Tests

We want to demonstrate the ability of our system to dynamically enforce security policies, the latter being consistent in spite of the migration of the nodes. In this chapter, we expose the tests we want our prototype to pass in order to consider that it corresponds to the requirements.

The first part will expose the scenarios that we imagined in order to cover all the use cases. The second part will expose the tools we chose to use to run these tests and the third part will present the created components in order to do so.

5.1.1 Scenarios

In order to prove that our system can answer the problems that we highlighted in this project, we need to create scenarios which are relevant. The scenarios must cover all the properties we want our model to have.

Explicitly, it means that our system must firstly secure the traffic by automatically enforce security policies. Then, those security policies must stay coherent although nodes are

migrating across the data center. Finally, all or part of the security policies can be delegated to another zone when the source and migration machines are not located in the same zone.

Scenario 1 - Dynamically Enforce Security Policies

There are several important words when we say that we want to *dynamically enforce security policies*. The first word is *dynamically*. It means that the policies are preset at the network level, and as a node is emitting a packet, the network controller will apply the adequate policy to the flow without any human intervention.

Another important notion is *security policies*. We have previously described what it means : a security policy corresponds to a chain of middlebox to be ensured. The actual traversal of these middleboxes must be verified. No other middlebox must be traversed.

Therefore, the first scenario will take place in a network across which are spread middleboxes. Two senders will send packets to a receiver. There will be a different security policy for each of the sender. The senders and the receiver of the communication will be located on two different physical servers.

Scenario 2 - Node Migration

The security policies applied to the traffic must be coherent even if the nodes are migrating. In this scenario, the two previous senders of the scenario 1 will migrate to another part of the data center, on other physical machines.

This scenario aims to prove the property of our model to remain consistent even after node migration.

Scenario 3 - Wildcard

In a context where the data center can be partitioned in several zones, each ruled by a specific controller, we identified the possibility of a case we called a wildcard case.

In this case, all or part of the security policy enforcement can be delegated to another zone.

Our scenario will require two Virtual Machines, located in two different zones. In our scenario, VM1 located in Zone1 will communicate with VM2, located in Zone2.

5.1.2 Tools

In order to realize these tests, we need to simulate a network, in which nodes are migrating, sending and receiving packets. Furthermore, some of the nodes of the network will be middleboxes.

We will first review the different ways existing to create a testbed for our network before studying what are the ways to simulate the communicating nodes.

For each subsection, we will review the possible solutions before explaining the reason of our choices.

The Network

In order to run our tests, we need a network supporting OF, controlled by our handcraft controller. The most important characteristic of this network is that the nodes will migrate. However, we feel that creating a testbed running an actual cloud operating system would not be a good idea, as there is no hypervisor providing the capability to insert the AppID into the packet, as the VM starts emitting.

When a new architecture has to be evaluated, the most usual way to proceed is to use a network simulator as ns-2/ns-3 or Opnet. These simulators are very powerful and crucial in order to gather precise measurements on the network behavior.

However, the modeling with such simulator is a very complex and time-consuming task.

As we only want to gather a proof of concept, we believe these simulators are too advanced to fit the best our requirements. It would be like killing a fly with a bazooka.

Mininet allows to simply create emulated software-defined networks. The creation of a topology can be done programatically. Furthermore, graphic terminals can be run on the nodes, on which scripts can easily be run.

We believe it is complete enough as it creates a virtual emulated network where we can interact directly with the nodes. Yet, it provides simplicity required to set our scenarios and run our tests in order to prove the behavior of our system. We believe it is a good trade-off between a simplicity of a simulation and the realism of a testbed.

The Nodes

Now that we know how we're going to build our network, we can tackle the problem of the nodes.

First of all, the nodes will have to migrate and yet, there is no migration mechanism in place in Mininet. However, there is the possibility to run scripts on the nodes of the network.

Therefore, we believe the smartest way to proceed is to script the sending and receiving of packets. That way, we can simply kill a script and launch it on another node in order to simulate a migration.

The presence of middleboxes is the second hurdle in our tests.

Our first thought was to use actual middleboxes security processes. However, the installation and the configuration of these processes would be an unnecessary effort as it would not provide any more realism to our model. Furthermore, the compatibility of these processes and their adaptation to our installation would be a loss of time.

A security middlebox has a simple behavior. It consists in receiving a packet, detecting threats based on preset criteria, and if the packet is considered harmless, resend the packet. From the network point of view, the only behavior we're interested in is the receiving and sending of a packet.

Therefore, we decide to emulate the behavior of a middlebox with a script. As the controller is coded in Python, we will create a Python script modeling the middlebox behavior.

All types of nodes are then covered and will be emulated by Python scripts, emitting and/or receiving packets.

5.1.3 Created Components

All the scenarios have been defined and the tools have been selected.

In this section, we will present the components we created in order to make possible the realization of the tests. We will first focus on the creation of a network topology before presenting the scripts we'll use to emulate the nodes and middleboxes.

The Network

In order to realize the three scenarios mentioned above, we need to create particular network topologies.

In order to simplify the tests, we think it's better to create a single topology, meeting the needs of the three scenarios all at once.

From the scenarios analysis, we first see that the topology must be divided in 2 zones, each ruled by a specific controller.

The first zone will be the scene of the two first scenarios. The first scenario requires the creation of the senders nodes and the receiver nodes. In the data center must lie middleboxes. In order to combine the two scenarios, we will create a symmetrical network where the traffic can go through the same type of middleboxes after migration, in order to demonstrate the consistency of a security policy despite the difference of middleboxes instances.

The third scenario is the use of a wildcard. The receiver of the communication established in the third scenario will therefore have to be in the second zone. Several middleboxes also have to be in the second zone in order to delegate the traversal of those middleboxes.

We use the Python API provided by Mininet in order to create the custom network

meeting the requirements mentioned above. The created topology is presented on the Figure 5.1. The code used to generate this topology can be found in the annex B.

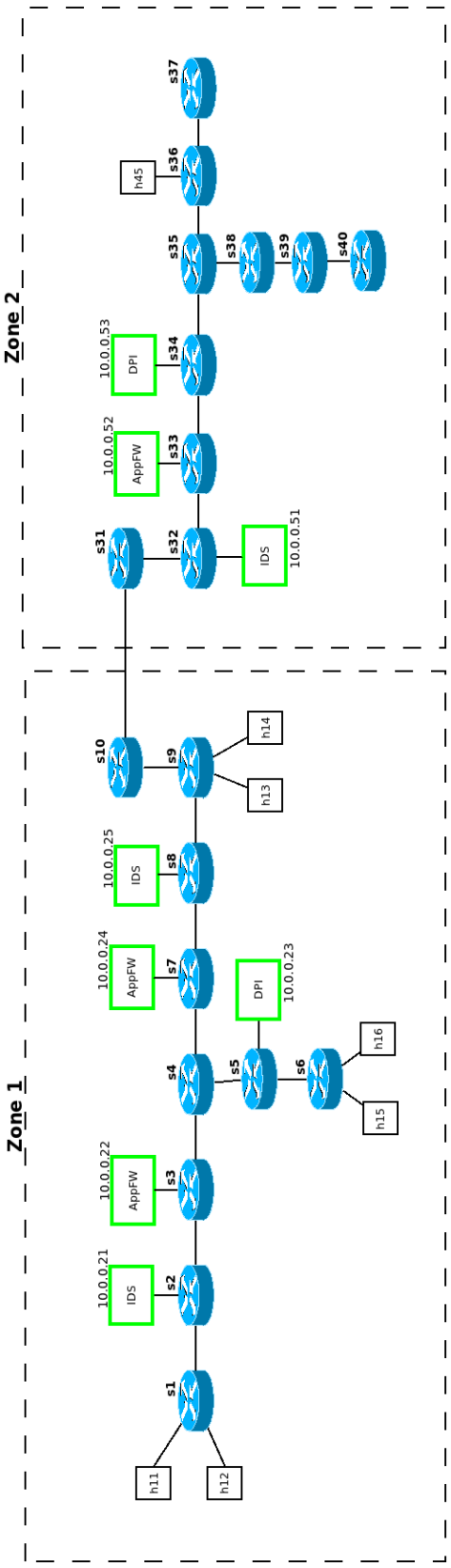

Figure 5.1 Topology of the test network

The Nodes

The scenarios are written, the network is set and the controller ready to operate. The only thing missing at this point is the nodes.

As we have explained at the section 5.1.2, the senders, receivers and middleboxes will be emulated by Python scripts.

The first script we implemented was the one designed to receive packets. It simply opens a socket and listens to incoming packets. When an IP packet is detected, the information regarding the source and the destination are print on the terminal.

The second script we implemented is the one designed to send packets.

An IP packet is crafted with the adequate information regarding the addresses of the sender and the receiver as well as the Application ID. A socket is then opened and the packet is sent out.

The third and last script we implemented is the one designed to emulate the behavior of middleboxes.

The script opens a socket and listens to incoming packets. When an IP packet is detected, the information regarding the source and the destination are printed out.

The unmodified packet is then re-sent out to the network.

We decided to use explicit names for our sending scripts, by creating one script per source-destination combination. For example, the script designed to send packets from the node h11 (10.0.0.11) to h15 (10.0.0.15) will be named *1115sendpackets.py*.

The code of *1115sendpackets.py* can be found in annex C. The code of *receivepackets.py* can be found in annex D. The code of *middlebox.py* can be found in annex E.

5.2 Results

As everything is now implemented, the tests can actually be realized. In this section, we'll analyze the three scenarios independently. For each of them, we'll describe the manipulations step by step. As the scenarios will unfold, we will describe the behavior of the system. Then, we will analyze the results of the tests for each scenario.

As the manipulations occur, we will record a video showing the behavior of the different elements of the network. In this thesis, we will present only captures of the video.

In this section, we will often refer to the topology found on the figure 5.1.

The name of the nodes on top of which runs the middlebox scripts are named after their IP address. For example, the IDS with the IP address 10.0.0.21 is the node h21.

All the flow entries at the OFS level are shown on the annex F.

5.2.1 Scenario 1 - Automatically Enforce Security Policies

Manipulation

This scenario occurs in Zone 1. We will simulate the communication between the nodes h11 and h15. This communication is marked with the AppID corresponding to the security chain *(IDS, AppFW)*.

We will also simulate the communication between the nodes h12 and h16. This communication is marked with the AppID corresponding to the security chain *(IDS, AppFW, DPI)*.

Observation

As the script *1115sendpackets.py* is ran on the node h11, the flow is instantly set across the switches of the data center. We can see that the middleboxes placed on h21 (IDS) and h22 (AppFW) are traversed. The packet is then received on the node h15, where the information of the packet is printed.

As the script *1216sendpackets.py* is ran on the node h12, the flow is instantly set across the switches of the data center. We can see that the middleboxes placed on h21 (IDS), h22 (AppFW) and h23 (DPI) are traversed. The packet is then received on the node h16, where the information of the packet is printed.

The behavior of the network after the script *1115sendpackets.py* has been run is shown on figure 5.2.

Analysis

We can see that in both cases, the security policy has been applied automatically, without a manual intervention. The middlebox sequences has been applied accordingly to the Application ID present in the packet.

5.2.2 Scenario 2 - Node Migration

We use the scenario 1 as a basis for this new scenario.

Manipulation

Once the communication is established, we simulate the migration of the node h11 to the location of the node h13 on the right side of zone 1. We will also simulate the migration of the node h12 to the location of the node h14.

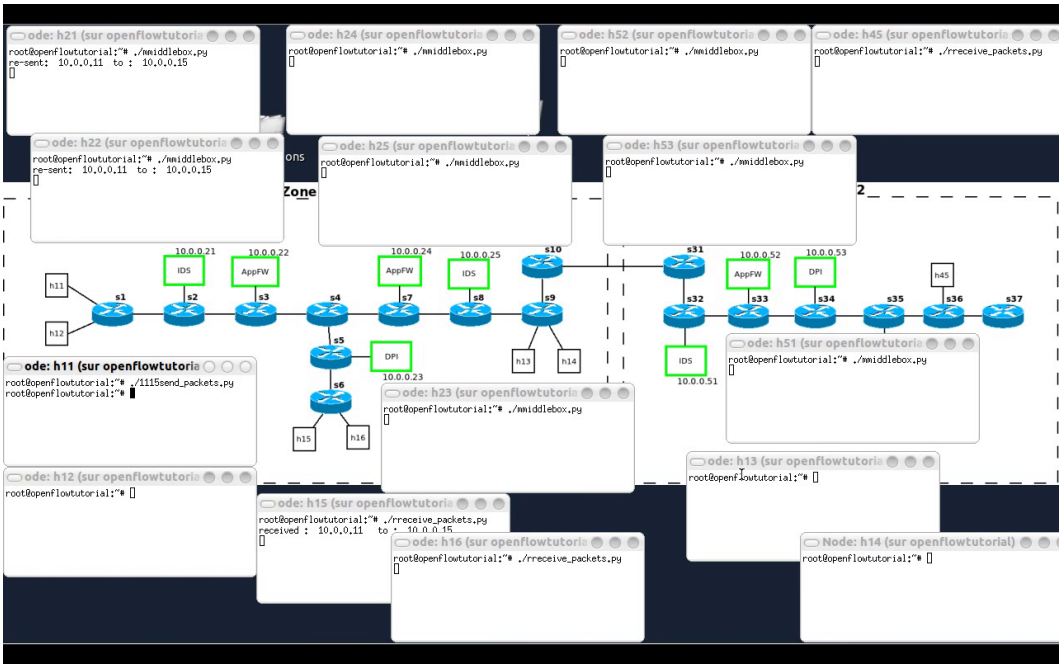

Figure 5.2 Network behavior after running *1115sendpackets.py*

Observation

As the node h11 migrates to h13 and starts emitting packets again, the flow is instantly set across the switches of the data center. We can see that the security policy is still applied *(IDS, AppFW)*. However, the instances that are traversed have changed. Before migration, the flow was going through *(h21, h22)* meanwhile the new flow is traversing *(h25, h24)*. The packet is still seamlessly received on the node h15.

The node h12 also migrates to h16. Similarly, before the migration, the flow was going through *(h21, h22, h23)* as the new flow is traversing *(h25, h24, h23)*. The packet is still seamlessly received on the node h15.

The behavior of the network after the migration of h11 in h13 and h12 in h14 is shown on figure 5.3.

Analysis

We can see that in both cases, the security policy has been reconfigured automatically after migration. The middlebox sequences has been applied accordingly to the Application ID present in the packet, even though the instances traversed have changed. We chose to show both these cases of migration in order to demonstrate that *all or part* of the traversed middlebox instances can be modified.

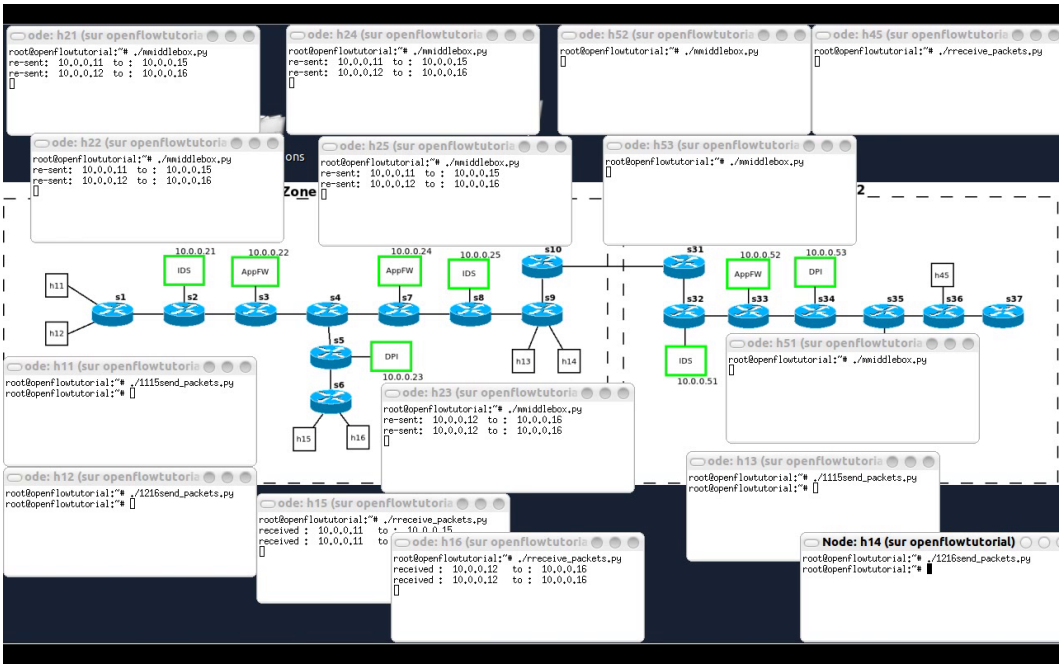

Figure 5.3 Network behavior after the migration of h11 and h12

We can conclude that our model ensure the consistency of the security policies even after the migration of a node.

5.2.3 Scenario 3 - Wildcard

In this scenario, we will establish a communication between two nodes located in two different zones of the data center, ruled by two different controllers.

Manipulation

We will simulate the communication between the nodes h12 and h45. The node h12 is located in Zone 1 as the node h45 is located in Zone 2, as shown on figure 5.1. This communication is marked with the AppID corresponding to the security chain *(IDS, AppFW, DPI)*.

Once the communication is established, we will migrate the node h12 from the location h12 to h14, as previous.

Observation

As the node h12 starts emitting packets, the flow is instantly set across the switches of the data center. We can see that the security policy is applied through the instances *(h21,*

h22, h53). The nodes h21 and h22 belong to Zone 1 but the traversal of the *DPI* has been delegated to the controller of the second zone.

As we migrate h12 to the location h14, we can see that the security policy is now applied through the instances *(h51, h52, h53)*. All three nodes h51, h52 and h53 belong to the Zone 2. In this case the controller of Zone 1 delegated all the enforcement of the security policy to the controller of Zone 2.

The application of the wildcard after the migration of h12 in h14 is shown on figure 5.4.

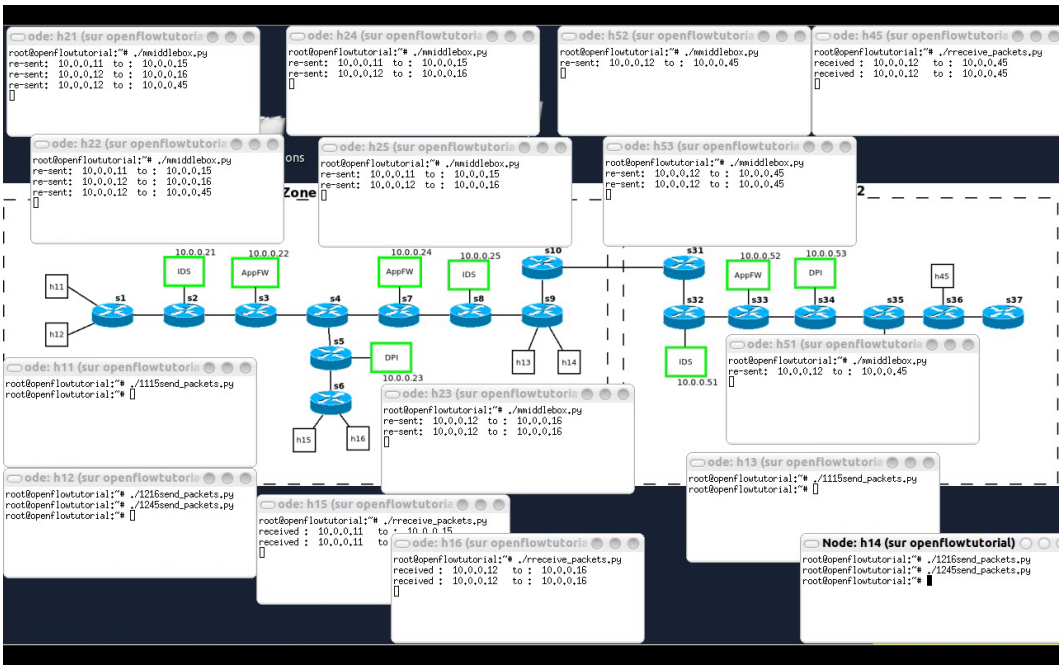

Figure 5.4 Network behavior after the migration of h12

Analysis

The third scenario shows that the application of a security policy can be applied even throughout several zones ruled by different controllers.

The middlebox sequences has been applied accordingly to the Application ID found in the packet, and this, in two distinct zones.

We chose to show the migration of the node h12 in order to demonstrate that *all or part* of the security policy enforcement can be delegated to another zone.

We can conclude that our model ensure the consistency of the security policies even throughout several zones.

5.3 Limitations and Future Work

Our model allows us to dynamically enforce security policies in a multi-tenant cloud network. In addition, the security policies stay coherent in spite of the node migration.

This chapter will focus on the limitations of the routing protocol of our implementation. These limitations correspond with the future work that can be done in order to improve our architecture.

5.3.1 Coherence of the Controllers

The first issue that has not been addressed in our project is the coherence of the controller. Indeed, one challenge is to implement a way to keep the same configuration on all the controllers, in order for all the nodes to be treated in a similar fashion.

This is one of the most important future research that we identify for our project.

5.3.2 Coherence of the Middlebox Configuration

Another unresolved issue concerns the coherence of middlebox configuration. Indeed, if modifications are made to the configuration of a middlebox, all middlebox of the same type will have to be modified simultaneously. If there is no mechanism to ensure this coherence, it may results in packets treated differently by the same middlebox type.

5.3.3 Traffic Overhead

In our thesis, we presented simple use cases. However, in the complexity of a real data center, the topology may be complex. In such a situation, the routing of the packets through the proper middleboxes may result in complex routes and leads to an important traffic overhead.

Such situations has not been studied and the potential overhead has not been measured.

5.3.4 Potential Deadlocks

As the security chain is retrieved at the controller level, it is translated into middlebox instances to traverse. In the case where there is no middlebox of a certain type, up and running in a zone of the data center, it may result in the impossibility to enforce the security policy. In this case, all packets would be dropped.

This case would not be tolerable in a real environment. Deadlock possibilities have not been studied during this project.

5.3.5 Routing Protocol

So as to create a prototype of our network controller, we had to implement several modules in order to demonstrate the relevance of our policy enforcement mechanism. Particularly, a routing protocol has been implemented in order to figure out the route from the source to the destination, throughout the middleboxes.

The security policy requires certain types of middleboxes to be traversed, in a particular order. However, there can be several instances running in the network, for a particular middlebox type.

Our routing protocol considers the possible routes throughout all the eligible combinations of instances. The number of hops of each route is calculated.

Such a mechanism is not optimized. Calculating all the possible routes can be time-consuming and affect the scalability of our system.

However, the prototype we presented is working on a simple and small topology as we only sought a proof of concept.

Therefore, in order to provide a better solution, the routing protocol will have to be further studied and optimized.

5.3.6 EEL-tag Implementation

The EEL-tags are used in order to route the packets across the network, inside the data center. Our model does not require a specific way to insert the iTags and gTags to the packets. However, as we implemented the solution, we had to choose a way to proceed.

We decided to use the VLAN tag id in order to insert our EEL-tags. The first version of the OpenFlow protocol requires for the switches to have the capability of pushing, popping or modifying VLAN tags.

The current implementation allows us to have more than a hundred middlebox types, and more than a hundred instances of each type.

If this number would not be high enough, the new OpenFlow protocol allows to push, pop and modify the MPLS header tag (20 bit long). We believe that it could provide enough scalability to our model, if needed.

5.3.7 Rule Optimization

The routing based on the EEL-tags enables the aggregation of flows, as several of them can be matched based on a unique rule, matching an unique tag.

However, no further mechanism has been put in place in order to optimize the flow-entries at the OF-switch level. Furthermore, the tagging of the flows is done at the middleboxes'

egress switches. Therefore, we believe the it is where the highest numbers of rules will be. However, we haven't evaluated the number of these rules. We haven't tried to optimize their number as well.

5.3.8 Resource Management

In a wildcard case, the controller of a zone has the possibility to delegate the traversal of middleboxes to another zone.

This decision can be made based on several criteria. First, the overhead caused by the detour of routing the flow through the middleboxes could be a determining factor. Indeed, we can easily imagine a case where sending the flow directly to another zone would be the shortest path. However, it would be difficult to evaluate how such a decision would be a good trade-off overall. Indeed, it is possible that the shortest path in the first zone would actually lead to the longest path, after all zones have been considered.

We believe that the wildcard is a good example that the routing based on the shortest path is not the most efficient. The selection of the paths could be done while operating a resource management of the middleboxes. Indeed, if the controller had an information on the load of the middleboxes, it would be able to allocate the flows equally between the middleboxes. This would enhance the scalability of the system.

Such a mechanism is out of scope and therefore has not been studied in our work. We nonetheless believe that it could be a field of interest for the future works.

5.4 Comparison to solutions from the literature

In the previous sections, we have built and analyzed our solution. We highlighted its limitations as well as the directions of future work. In the current section, we will compare our solution to the most relevant ones found in the literature.

In order to do so, we compare them by determining whether they meet criteria key to the enforcement of security policies in multi-tenant networks. Regarding the solutions of the literature, we studied their properties in the literature review.

The first criteria we identified is the need for the traffic to be routed through a chain of middleboxes, or at least one middlebox. The proof of concept clearly shows that the traffic is routed through chains of middleboxes.

Furthermore, the traffics of the different tenants has to be isolated from each other. As our solution builds on Software-Defined Network, the isolation is done as the flows are matched based on characteristics such as the source and destination addresses.

Scalability is another key requirement for any system operating in a cloud environment.

As it is difficult to evaluate the actual numbers of nodes in a cloud environment, we provide scalability by allowing the data center to be partitioned in smaller zones, each zone being ruled by a particular controller. The proof of concept showed how the interaction between several zones are made, by delegating the enforcement of part or all of a policy to another zone.

One of the key issues we identified is the need for the enforcement of policies to be done automatically. Indeed, the size of the data centers don't allow a manual enforcement, as it would cost too much time and manpower. The proof of concept showed that as the packets are sent, the enforcement of the security policies is done automatically, without any human intervention.

The tolerance to migration is another key aspect of any system operating in the cloud. Our proof of concept proves that upon a node's migration, the path is recalculated and enforced in order to provide the same middlebox chain, even though it could be applied using different middlebox instances.

A comparison of the algorithms can be found in table 5.1.

Table 5.1 Comparison of algorithms

requirement / Solution	Middlebox	isolation	scalability	automatism	migration
Policy-aware [joseph]	Yes	Yes	No	Yes	No
NetOdessa [bellessa]	No	Yes	Yes	Yes	No
FML/FSL [hinrichs]	Yes	Yes	Yes	No	No
Secure cloud [hao]	Yes	Yes	No	No	No
Our Solution	Yes	Yes	Yes	Yes	Yes

CHAPTER 6

CONCLUSION

All good things come to an end, and this project is no exception.

As we enter the last chapter of this work, we are going to review all the progress that has been done since the enunciation of the problem we tackled in this project, the dynamic enforcement of security policies in multi-tenant cloud networks.

Applying security policies is not new, but the cloud brings on new requirements that we must meet. We started our report with the review of the literature. Some research has addressed the issue of network security by developing solutions enabling the traversal of middleboxes.

However, no solution succeeds to provide the consistency of the security policies despite the node migration.

6.1 The Research Project

We developed a model based on Software-Defined Network, and particularly OpenFlow, the most advanced protocol in this area to date. A centralized controller will operate on a set of switches. Those switches will send to the controller any packet belonging to a new flow.

As the controller receives this packet, it retrieves the AppID inserted in the packet by the hypervisor. This AppID corresponds to a security policy, in the form of a chain of middlebox type, represented by gTags. This chain of middleboxes is then translated in iTags, each iTag corresponding to an actual running middlebox instance of the network.

The controller then identifies the path from the source to the destination and throughout the selected middlebox instances. The routing rules are pushed down to all forwarding elements along the path.

The iTags are set by the switches to the packets, in order for them to be routed to the adequate middleboxes. The use of those tags allows us to aggregate many flows with one rule.

This mechanism allows us to reconfigure the rules upon each migration of the nodes. As the migrated machine starts emitting packets again, the same process as previous is followed, and the security policy is applied. The enforcement of the security policy may be done through different middlebox instances from the ones traversed prior to the migration.

In order to improve the scalability of our model, we allow the data center to be partitioned in smaller zones, each ruled by one controller. Our controller design allows the security policy to be applied throughout several zones, as the controller of the first zone can delegate the application of all or part of the middlebox chain to the next zone controller.

We further built a prototype of our solution, in order to prove the viability of our concept.

We imagined several scenarios so as to cover all the situations we believe are relevant to our project.

Our tests firstly prove that the enforcement of the security is done automatically, without any human intervention. The middlebox sequence traversed corresponds to the one defined in the security policies.

When a node migrates, our second test demonstrates that the security policy is still applied after the migration. Indeed, the paths are reconfigured as the VM starts emitting packets again. The enforcement of the security policy can be realized through different instances of the middleboxes.

The last test we implemented demonstrates the wildcard mechanism. It proves that the enforcement of the security policy can be achieved throughout several zones, ruled by distinct controllers.

6.2 Limitations

We designed a mechanism in order to ensure the consistency of the security policies in spite of the migrations of the nodes. Our implementation is not designed with any performance requirements, as the complete system is not only depending on our mechanism but mostly to the route calculation module.

The routing of the packets is implemented in a very simple way and we believe that future work should focus on the optimization of the routing rules. Indeed, we believe it is the key element in order to create an architecture with good enough performance to be implemented in a production cloud.

Furthermore, the tests have been implemented in a simple network. Our model could suffer traffic overhead under a complex topology. This aspect has not been studied in the project.

Moreover, a distributed architecture could be very difficult to manage, as there is no way to maintain the coherence between the controllers.

6.3 Future work

Future work should focus on the wildcard mechanism and the resource management possibilities that it offers. Indeed, the delegation of middlebox traversal to another data center imposes to have rules on which to make the decision on the delegation. Our prototype does it in a very simple way, as it only chooses the shortest path among the different routing possibilities.

We believe that it would be interesting to balance the load between the different middleboxes, thus enhancing the scalability of our solution.

I would also important to provide a mechanism to keep the configuration coherent between the controllers. A similar mechanism could be implemented for the middleboxes.

All good things come to an end, and this conclusion paragraph is no exception.

This project contributes to the security of the cloud at the network level. The focus of the whole community is today on the cloud, as it is a rising technology. The next generation will use computers in a way we can only dream about for the moment.

Few are the visionaries able to predict what the future of computing will be. However, we believe the brighter days are still to come for cloud computing.

REFERENCES

[1] Resource description framework. ResourceDescriptionFramework, 2004.

[2] Amazon EC2 Dedicated Instances. http://aws.amazon.com/fr/dedicated-instances/, 2011.

[3] AWS GovCloud. http://aws.amazon.com/fr/govcloud-us/, 2011.

[4] Dedicated cloud and server. http://www.rackspace.com/cloud/public/servers/?cm_mmc=PPCCloudBU-_-Google-_-broad-_-dedicated+cloud+server, 2011.

[5] Govcloud. http://govcloud.com/, 2011.

[6] Ieee 802.1q-in-q vlan tag termination. http://www.cisco.com/en/US/docs/ios/lanswitch/configuration/guide/lsw_ieee_802.1q.html, 2011.

[7] Report: Google Uses About 900,000 Servers. http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers, 2011.

[8] Sony estimates 171 million dollars cost for psn breach. http://www.engadget.com/2011/05/23/sony-estimates-3-2b-loss-this-year-171-million-cost-for-psn-b, 2011.

[9] Aeisha Bright, Ramesh Isaac, Alex Nadimi, Chris O'Brien, Chris Reno, Henry Vail, Mike Zimmerman, Serge Maskalik, Wen Yu. *Enhanced Secure Multi-Tenancy Design Guide*. Cisco, October 2010.

[10] John Bellessa, Evan Kroske, Reza Farivar, Mirko Montanari, Kevin Larson, and Roy H. Campbell. Netodessa: Dynamic policy enforcement in cloud networks. In *Proceedings of the 2011 IEEE 30th Symposium on Reliable Distributed Systems Workshops*, SRDSW '11, pages 57–61, University of Illinois at Urbana-Champaign, 2011. IEEE Computer Society.

[11] Ben Pfaff, Bob Lantz, Brandon Heller, Casey Barker, Dan Cohn. *OpenFlow Switch Specification*. Open Networking Foundation, February 2011.

[12] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[13] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, nov. 2008.

[14] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: an open platform for network research. *SIGCOMM Comput. Commun. Rev.*, 33(1):53–57, January 2003.

[15] Fang Hao, T. V. Lakshman, Sarit Mukherjee, and Haoyu Song. Secure cloud computing with a virtualized network infrastructure. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.

[16] I. Hareesh, S. Prasanna, M. Vijayalakshmi, and S.M. Shalinie. Anomaly detection system based on analysis of packet header and payload histograms. In *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*, pages 412 –416, june 2011.

[17] Tim Hinrichs, Natasha Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Expressing and enforcing flow-based network security policies, November 2008.

[18] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM.

[19] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):51–62, August 2008.

[20] Ali Khajeh-Hosseini, David Greenwood, and Ian Sommerville. Cloud migration: A case study of migrating an enterprise it system to iaas. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 450–457, Washington, DC, USA, 2010. IEEE Computer Society.

[21] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[23] Kevin Larson Wucherl Yoo Roy H. Campbell Mirko Montanari, Ellick Chan. Distributed security policy conformance. In *SEC*, IFIP Advances in Information and Communication Technology, pages 210–222, Washington, DC, USA, 2011. Springer.

[24] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. *SIGCOMM Comput. Commun. Rev.*, 41(4):62–73, August 2011.

[25] Network Strategy Partners. *MANAGEMENT CONSULTANTS TO THE NETWORKING INDUSTRY*. Network Strategy Partners, LLC., March 2007. www.nspllc.com.

[26] Maureen Rogers. State of cloud security report, May 2012.

[27] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[28] Voorsluys, William and Broberg, James and Buyya, Rajkumar. *Cloud Computing*. John Wiley and Sons, Inc., 2011.

APPENDICES A

controller.py

```
# EEL Controller
# Zone 1

import logging
import array
import sys
#import pcap
import string
import time
import socket
import struct

import nox.lib.openflow as openflow

from struct import *
from nox.lib.core import *
from nox.lib.packet.ethernet import ethernet
from nox.lib.packet.packet_utils import mac_to_str, mac_to_int,
    ipstr_to_int
from nox.netapps.topology.pytopology import *
from nox.netapps.authenticator.pyflowutil import Flow_in_event
from nox.netapps.routing import pyrouting
from nox.netapps.discovery.discovery import *
from nox.lib.netinet import netinet
from nox.coreapps.pyrt.pycomponent import CONTINUE

log = logging.getLogger('nox.coreapps.tutorial.pytutorial')


class pytutorial(Component):
```

```
def __init__(self, ctxt):
Component.__init__(self, ctxt)
self.routing = None
self.discovery = None

# This table is created in order to keep track of hosts connected
    to switches.
# it is initialize empty
self.mac_ip_to_switch_port = {}    # key: (IP addr); value: (dpid,
     port)

# example pour EEL-topo : {(13L, 167772173): (9L, 2), (4L,
    167772164): (10L, 3), (3L, 167772163): (10L, 2), (12L,
    167772172): (8L, 2), (2L, 167772162): (5L, 2), (1L, 167772161):
    (5L, 1), (11L, 167772171): (7L, 2)}


# this table links the EEL-gTag to the EEL-iTag range to which it
    corresponds
self.gtags = {
1: (128,255), # DPI
2: (256,383), # AppFW
3: (384,513), # IDS
4: (514,639), # IPS
5: (640,767)  # FW
} #gtag <-> itag range

# this table links the Application IDs (AppIDs) to the security
    chains they represent
self.app_id = { #  dans la chaine l'indice commence a 0
1: (1,2,3), # dpi appfw ids
2: (3,2),   # ids appfw
3: (1,2,5), # dpi appfw fw
4: (1,4,5), # dpi ips fw
5: (2,3,5), # appfw ids fw
6: (3,5),   # ids fw
```

```
7: (3,2,1), # ids appw dpi
8: (2,5)     # appfw fw
} #app_id <-> gtag chain

# this table links the EEL-iTags to the MAC and IP address of the
    security appliance
self.itags = {
421: (1,"10.0.0.21"), # IDS
425: (1,"10.0.0.25"), # IDS
322: (1,"10.0.0.22"), # AppFW
324: (1,"10.0.0.24"), # AppFW
223: (1,"10.0.0.23")  # DPI
} #itag <-> (mac,ip adress)


self.gateway = {
"10.0.0.45": (10L,2), # 10.0.0.45 is in the zone linked on the
    port 2 of switch 10
} #itag <-> (mac,ip adress)


# this function is called when an unregistered host is discovered.
# The MAC and IP addresses of the new hosts are stored in the
    table, linked to the datapath ID and its port through which the
    packet arrived
def learn(self, ip_addr, dpid, inport):
self.mac_ip_to_switch_port[ip_addr] = (dpid,inport)

# used to register ip addresses of hosts from other regions
# The IP address is linked to the gateway datapath
def learn_gateway(self, ip_addr, dpid, inport):
self.gateway[ip_addr] = (dpid,inport)


# function : checks for an EEL-tag on the packet, in case of a
    wildcard case incoming from another region
```

```
def check_eel_tag(self, buf):
#is_tagged = buf.tolist().find('mpls')
return 0

# function : returns the security chain (gTag chain) from a given
    AppID
def retrieve_chain(self, app_id):
chain = self.app_id[int(app_id)]
return chain



# obsolet
def learn_and_forward(self, dpid, inport, packet, buf, bufid):
# Initial hub behavior: flood packet out everything but input port
    .
self.send_openflow(dpid, bufid, buf, openflow.OFPP_FLOOD, inport)


# for a given MAC,IP couple, returns 1 if it's in the host table,
    0 otherwise (used to check if a wildcard is possible)
def is_dst_in_table(self,ip) :
if ip in self.mac_ip_to_switch_port.keys():
return 1
else:
return 0

# returns the (DPID,PORT) switch associated to a host, based on it
    's MAC,IP addresses
def get_switch(self,ip) :
return self.mac_ip_to_switch_port[ip]

# return all the info of the packet    : [register, has_app_id,
    s_addr, d_addr, app_id, eel_tag]
def get_info(self, packet) :
register = 0
```

```
has_app_id = 0
s_addr = 0
d_addr = 0
ip_opt = 0
eel_tag = 0
#parse ethernet header
eth_length = 14
head_length = eth_length
eth_header = packet[:eth_length]
eth = unpack('!6s6sH', eth_header)
eth_protocol = socket.ntohs(eth[2])
#print "eth protocol ", eth_protocol
#Parse ARP packets
if eth_protocol == 1544 :
register = 1
arp_protocol = str(packet[16]) + str(packet[17])
if arp_protocol != "80" :
print "no_ip"
s_addr = str(packet[28]) + "." + str(packet[29]) + "." + str(
    packet[30]) + "." + str(packet[31])

# Parse VLAN header
if eth_protocol == 129 :
register = register + 2
vlan_length = 4
vlan_header = packet[eth_length:vlan_length+eth_length]
vlanh = unpack('!HH', vlan_header)
eel_tag = vlanh[0]
eth_protocol  = socket.ntohs(vlanh[1])
head_length = head_length + vlan_length


if eth_protocol == 8 :
#Parse IP header
#take first 24 characters for the ip header (classic header +
    options)
```

```
ip_header = packet[head_length:24+head_length]
#now unpack them :)
iph = unpack('!BBHHHBBH4s4sBBBB' , ip_header)
version_ihl = iph[0]
version = version_ihl >> 4
ihl = version_ihl & 0xF
iph_length = ihl * 4
ttl = iph[5]
protocol = iph[6]
# if it's not UDP
if protocol != 1 :
# ignoring ICMP packets
register = register + 1
has_app_id = 1
s_addr = socket.inet_ntoa(iph[8]);
d_addr = socket.inet_ntoa(iph[9]);
ip_opt = iph[10]
return [register, has_app_id, s_addr, d_addr, ip_opt, eel_tag]

# calculate the routes based on the segments
def find_route(self, segments) :
all_my_routes = list()
for segment in segments:
# the route represents the linkage of dpid/port along a segment
myroute = list()
forbidden_routes = list()
first  = segment[0]
next = first
last = segment[1]
route_incomplete = 1
route_possible = 1
go_on = 1
forbidden = 0
no_match = 1
bug_in_route = 0
while go_on :
```

```
route_possible = 1
while route_incomplete & route_possible :
for row in self.discovery.adjacency_list:
no_match = 1
forbidden = 0
if row[0] == next:
# check if the next hop isn't already in the route
for seg in myroute :
if seg[0] == row[2]:
forbidden = 1
for forbidden_route in forbidden_routes:
if (forbidden_route[0] == next) & (forbidden_route[1] == row[2]) :
forbidden = 1
if forbidden != 1 :
no_match = 0
myroute.append((row[0],row[1]))
if row[2] == last:
route_incomplete = 0
go_on = 0
break
else :
next = row[2]
break
if row[2] == next:
for seg in myroute :
if seg[0] == row[0]:
forbidden = 1

for forbidden_route in forbidden_routes:
if (forbidden_route[0] == next) & (forbidden_route[1] == row[0]) :
forbidden = 1
if forbidden != 1 :
no_match = 0
myroute.append((row[2],row[3]))
if row[0] == last:
route_incomplete = 0
```

```
go_on = 0
break
else :
next = row[0]
break
if no_match :
route_possible = 0
if len(myroute) == 0 :
print "*** ERROR : no route found"
go_on = 0
bug_in_route = 1
else :
forbidden_routes.append((myroute[len(myroute) -1][0],next))
myroute = list()
next = first

all_my_routes.append(myroute)
forbidden_routes = list()
return all_my_routes

# return the ingress switch based on the middlebox instance itag
def get_mid_switches(self, itag_instances) :
mid_switches = list()
# if only one element
if type(itag_instances) == type(tuple()) :
mid_switches.append(self.get_switch(ipstr_to_int(itag_instances
    [1])))
else :
for sw in itag_instances :
mid_switches.append(self.get_switch(ipstr_to_int(sw[1])))

return mid_switches


# parameters : the itag chain, the src and dst switches
# create and return the segments of route
```

```
def get_segments(self, mid_switches, src_switch, dst_switch) :

# we split the entire route from src host to dst host in sub-
    segment, src host to 1st middlebox, 1st middlebox to 2nd....
segments = list()

# if there is no middleboxes to traverse, only one segment
if len(mid_switches) == 0 :
segments.append((src_switch[0],dst_switch[0]))
# when middleboxes need to be traversed, we determine the segments
else :
# src host -> 1st middlebox
segments.append((src_switch[0],mid_switches[0][0]))
# between middleboxes
if len(mid_switches) > 1 :
for i in range(0,len(mid_switches) - 1) :
segments.append((mid_switches[i][0],mid_switches[i+1][0]))
# last middlebox -> dst host
segments.append((mid_switches[len(mid_switches)-1][0],dst_switch
    [0]))

return segments

# default function called when the packet_in event is raised
def packet_in_callback(self, dpid, inport, reason, length, bufid,
    packet):
"""Packet-in handler"""
if not packet.parsed:
log.debug('Ignoring incomplete packet')
else:
packet_info = self.get_info(packet.arr)
# we learn the host<->switch link
if (packet_info[0]==1):
src_ip  = packet_info[2]
# we register the new packet in the host<->switch table
self.learn(ipstr_to_int(src_ip) ,dpid,inport)
```

```
print "packet registered. table :", self.mac_ip_to_switch_port

# we learn the host<->gateway switch link
if (packet_info[0] == 3):
src_ip  = packet_info[2]
# we register the new packet in the host<->switch table
self.learn_gateway(src_ip ,dpid,inport)
print "gateway registered. table :", self.gateway

# we test if the encapsulated received packet is an IP packet,
   otherwise we just register it in the table
# if the packet contains an App_IP :
if (packet_info[1]):
src_ip = packet_info[2]
dst_ip = packet_info[3]
app_id = packet_info[4]

# variable representing the source datapath
src_switch = (dpid, inport)

# determine the gtag security chain
security_chain = self.retrieve_chain(app_id)

# determine whether or not the packet comes from another region
   and tagged
from_wildcard = packet_info[5]

# trim the security chain when the packets has been wildcarded in
   another region
if (from_wildcard != 0) :
new_chain = list()
is_in_chain = 0
# for each element in the security chain, we check if it
   corresponds to the EEL_tag found on the wildcard packet
# coming from another region. The new security chain begins at
   this gtag
```

```
for gtag in security_chain :
if gtag == from_wildcard :
is_in_chain = 1
if is_in_chain :
new_chain.append(gtag)
security_chain = new_chain

# checks if it's possible to do a wildcard
is_dst_reachable = self.is_dst_in_table(ipstr_to_int(dst_ip))


itag_ranges = list()
itag_instances = list()
itag_instances_combinations = list()
# if the destination is known, we get the ingress switch of the
    destination machine
if is_dst_reachable == 1 :
dst_switch = self.get_switch(ipstr_to_int(dst_ip))
# determiner les instance a traverser, faire un itag_chain
for gtag in security_chain:
itag_ranges.append(self.gtags[gtag])

# we determine the actual instances to be traversed
for i in range(len(itag_ranges)):
available_instances = list()
j = 0
# we get the available instances for a given middlebox type
for itag in range(itag_ranges[i][0],itag_ranges[i][1]):
if itag in self.itags.keys() :
instance_info = self.itags[itag]
if instance_info[0] : # the available bit
available_instances.append((itag,instance_info[1])) # the ip
    address
j = j + 1
```

```
L = len(itag_instances_combinations)
# if there is more than one available instances, we multipliate
    the items in the itag_instances_combination
# in order to create all the combinations of available instances
if j > 1 :
for k in range(len(available_instances)-1):
for n in range(L): # we demultipliate the chains
itag_instances_combinations.append(itag_instances_combinations[n])
# we now add the available instances to the chain containing the
    instances corresponding of the previous gtags
if L > 0 :
for k in range(len(itag_instances_combinations)) :
n = abs(k/L)  # used to determine which instance to append on
    which line
temp_line = list()
temp_line.append(itag_instances_combinations[k])
temp_line.append(available_instances[n])
itag_instances_combinations[k] = temp_line
else :
# liste vide, on met l'element
for instance in available_instances :
#temp = list()
#temp.append(instance)
#itag_instances_combinations.append(temp)
itag_instances_combinations.append(instance)
else :
if L > 0 :
for k in range(len(itag_instances_combinations)) :
temp = list()
if type(itag_instances_combinations[k]) == type(tuple()) :
temp.append(itag_instances_combinations[k])
else:
for elem in itag_instances_combinations[k] :
temp.append(elem) # sinon : on append dans la ligne k en crean
    just une ref
temp.append(available_instances[0])
```

```
itag_instances_combinations[k] = temp

else :
itag_instances_combinations.append(available_instances[0])

print "itag instances combinations .... ",
    itag_instances_combinations

# the destination is in a region ruled by another controller.
# we get the gateway switch which access the other region
if is_dst_reachable == 0 :
print " destination not reachable "
dst_switch = self.gateway[dst_ip]
for gtag in security_chain:
itag_ranges.append(self.gtags[gtag])

# we determine the actual instances to be traversed
for i in range(len(itag_ranges)):
available_instances = list()
j = 0
# we get the available instances for a given middlebox type
for itag in range(itag_ranges[i][0],itag_ranges[i][1]):
if itag in self.itags.keys() :
instance_info = self.itags[itag]
if instance_info[0] : # the available bit
available_instances.append((itag,instance_info[1])) # the ip
    address
j = j + 1


L = len(itag_instances_combinations)

if L == 0 :
for inst in available_instances :
temp = list()
temp.append(inst)
```

```
itag_instances_combinations.append(temp)
else :
max_len = 0
for elem in itag_instances_combinations:
if len(elem) > max_len :
max_len = len(elem)
for inst in available_instances :
for k in range(L) :
if len(itag_instances_combinations[k]) == max_len :
temp = list()
for elem in itag_instances_combinations[k] :
temp.append(elem) # sinon : on append dans la ligne k en crean
    just une ref
temp.append(inst)
itag_instances_combinations.append(temp)
```

```
# first we calculate the reference
itag_instances = itag_instances_combinations[0]
print "ref itag instances .... ", itag_instances
```

```
# when there is a wildcard possibility, the ref route is the one
    without any middlebox
if is_dst_reachable == 0 :
print "trying get mid with itag empty"
itag_instances = []

# determine the switches to whitch are attached the middleboxes
# this variable represents the DPID and PORT linked to the
    middleboxes
mid_switches = self.get_mid_switches(itag_instances)
```

```
segments = self.get_segments(mid_switches, src_switch, dst_switch)

# now for each segment
# we keep track of the current segment through the segment_index
    variable
segment_index = 0
final_actions = list()

the_routes = self.find_route(segments)


longueur_ref = 0
for route in the_routes:
longueur_ref = longueur_ref + len(route)

# then we check the lengths of the other routes and replace if
    necessary
for itag_chain in itag_instances_combinations :
# determine the switches to whitch are attached the middleboxes
# this variable represents the DPID and PORT linked to the
    middleboxes
mid_switches_temp = self.get_mid_switches(itag_chain)
segments_temp = self.get_segments(mid_switches_temp, src_switch,
    dst_switch)
the_routes_temp = self.find_route(segments_temp)

longueur_route = 0
for route in the_routes_temp:
longueur_route = longueur_route + len(route)

if longueur_route < longueur_ref :
longueur_ref = longueur_route
itag_instances = itag_chain
the_routes = the_routes_temp
```

```
mid_switches = mid_switches_temp
segments = segments_temp

if longueur_route == longueur_ref :
if len(itag_chain) > len(itag_instances) :
longueur_ref = longueur_route
itag_instances = itag_chain
the_routes = the_routes_temp
mid_switches = mid_switches_temp
segments = segments_temp



print "*** itag instances ..... ", itag_instances
print "*** calculated final routes : " , the_routes
print "*** route final length : " , longueur_ref


to_wildcard = [0 , 0]
# we get the gateway switch which access the other region
if is_dst_reachable == 0 :
gtag_index = len(itag_instances)
if gtag_index < len(security_chain) :
to_wildcard = [1, security_chain[gtag_index]]
print "to wildcard : ", to_wildcard

# set rules
for segment in segments:
print "segment index ", segment_index
print "the segment : " , segment
myroute = the_routes[segment_index]
print "myroute : " , myroute
nb_middlebox = 0
# variable representing the number of middleboxes
if (type(itag_instances) == type(tuple())) & (len(itag_instances)
   == 2) :
```

```
nb_middlebox = 1
else :
nb_middlebox = len(itag_instances)

# for the last segment : last middlebox -> dst host ... no more
    EEL_tags are required
######################################## todo : if wildcard, EEL
    tag present and routing accordingly
if (segment_index +1) > nb_middlebox :
for i in range(0,len(myroute)):
# each switch needs to route the packet based on the source and
    destination
attrs = {}
attrs[core.DL_SRC] = packet.src
attrs[core.DL_DST] = packet.dst
# for the first switch on the segment, we check the source port as
    well, except there is no middlebox traversed
if (i == 0) & (len(mid_switches) != 0):
attrs[core.IN_PORT] = mid_switches[segment_index-1][1]

if (i == 0) & (to_wildcard[0]):
print "put_wildcard_tag"
actions = [[openflow.OFPAT_SET_VLAN_VID, to_wildcard[1] ],[
    openflow.OFPAT_OUTPUT, [0, myroute[i][1]]]]
## if no mid switches has been traversed we need to set the final
    actions
if (len(mid_switches)==0) :
print "final_actions_put_wildcard_tag"
final_actions = [[openflow.OFPAT_SET_VLAN_VID, from_wildcard],[
    openflow.OFPAT_OUTPUT, [0, myroute[i][1]]]]
else :
actions = [[openflow.OFPAT_OUTPUT, [0, myroute[i][1]]]]
dp_id = myroute[i][0]
# match src dst : route
self.install_datapath_flow( dp_id, attrs, 60, 60, actions,
    buffer_id=None, priority=openflow.OFP_DEFAULT_PRIORITY, inport=
```

```
    None, packet=None)
print "set rule to dpid ", dp_id
print "and route outport", myroute[i][1]
print "i ", i

# for the last switch :
attrs = {}
attrs[core.DL_SRC] = packet.src
attrs[core.DL_DST] = packet.dst
actions = [[openflow.OFPAT_OUTPUT, [0, dst_switch[1]]]]
dp_id = dst_switch[0]
# match src dst : route to final machine
self.install_datapath_flow( dp_id, attrs, 60, 60, actions,
    buffer_id=None,priority=openflow.OFP_DEFAULT_PRIORITY, inport=
    None, packet=None)
print "set rule to last dpid ", dp_id
print "outport", dst_switch[1]
print "i ", i

# for any segment that ends with an middlebox ingress switch
else :
# determine the eel_tag to apply
if type(itag_instances) == type(tuple()) :
eel_tag = itag_instances[0]
else :
eel_tag = itag_instances[segment_index][0]
for i in range(0,len(myroute)):
# first switch of the segment needs to put the EEL_tag AND route
    the packet
if i == 0:
if segment_index == 0 :
if (from_wildcard != 0):
final_actions = [[openflow.OFPAT_STRIP_VLAN, 0],[openflow.
    OFPAT_SET_VLAN_VID, eel_tag],[openflow.OFPAT_OUTPUT, [0, myroute
    [i][1]]]]
else :
```

```
final_actions = [[openflow.OFPAT_SET_VLAN_VID, eel_tag],[openflow.
    OFPAT_OUTPUT, [0, myroute[i][1]]]]
attrs = {}
attrs[core.DL_SRC] = packet.src
attrs[core.DL_DST] = packet.dst
if segment_index != 0 :
attrs[core.IN_PORT] = mid_switches[segment_index-1][1]
# if the packet is from wildcard, we match the tag and pop it
if (from_wildcard != 0) & (segment_index == 0):
print "strip vlan id"
attrs[core.DL_VLAN] = from_wildcard
actions = [[openflow.OFPAT_STRIP_VLAN, 0],[openflow.
    OFPAT_SET_VLAN_VID, eel_tag ],[openflow.OFPAT_OUTPUT, [0,
    myroute[i][1]]]]
else:
actions = [[openflow.OFPAT_SET_VLAN_VID, eel_tag ],[openflow.
    OFPAT_OUTPUT, [0, myroute[i][1]]]]
dp_id = myroute[i][0]
# match from src to dst : insert the EEL_TAG
self.install_datapath_flow( dp_id, attrs, 60, 60, actions,
    buffer_id=None,                    priority=openflow.
    OFP_DEFAULT_PRIORITY, inport=None, packet=None)
print "set rule to dpid ", dp_id
print "from src to dst. put eel_tag ", eel_tag
print "and route outport", myroute[i][1]
print "i ", i

# other switches of the segment needs to route the packet based on
    the EEL tag
else :
# all other switches needs to route the packet based on the EEL
    tag
attrs = {}
attrs[core.DL_VLAN] = eel_tag
actions = [[openflow.OFPAT_OUTPUT, [0, myroute[i][1]]]]
dp_id = myroute[i][0]
```

```
# match EEL tag : route
self.install_datapath_flow( dp_id, attrs, 60, 60, actions,
    buffer_id=None,                    priority=openflow.
    OFP_DEFAULT_PRIORITY, inport=None, packet=None)
print "set rule to dpid ", dp_id
print "match eel tag ", eel_tag
print "and route outport", myroute[i][1]
print "i ", i

# last switch of the segment needs to pop the EEL tag
# also needs to route towards the middlebox
attrs = {}
attrs[core.DL_VLAN] = eel_tag
actions = [[openflow.OFPAT_STRIP_VLAN, 0],[openflow.OFPAT_OUTPUT,
    [0, mid_switches[segment_index][1]]]]
dp_id = mid_switches[segment_index][0]
# match eel_tag : pop the EEL_TAG and route through appropriate
    port
self.install_datapath_flow( dp_id, attrs, 60, 60, actions,
    buffer_id=None, priority=openflow.OFP_DEFAULT_PRIORITY, inport=
    None, packet=None)
print "set rule to dpid ", dp_id
print "pop eel tag ", eel_tag
print "outport", mid_switches[segment_index][1]
print "i ", i


segment_index =segment_index + 1



# at the end we must send the packet to the first switch, because
    it was sent to the controller and its processing needs to be
    resumed

self.send_openflow(dpid, bufid, packet, final_actions, inport)
```

```
return CONTINUE

def install(self):
self.routing = self.resolve(pyrouting.PyRouting)
self.discovery = self.resolve(discovery)
self.register_for_packet_in(self.packet_in_callback)

def getInterface(self):
return str(pytutorial)

def getFactory():
class Factory:
def instance(self, ctxt):
return pytutorial(ctxt)

return Factory()
```

APPENDICES B

EEL-topo.py

```
#!/usr/bin/python
"""
This example creates a multi-controller network for the
   demonstration of the EEL-controller.
by RaaverooK
"""
from mininet.net import Mininet
from mininet.node import Controller, OVSKernelSwitch,
   RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel

Switch = OVSKernelSwitch

def addHost( net, N ):
"Create host hN and add to net."
name = 'h%d' % N
ip = '10.0.0.%d' % N
return net.addHost( name, ip=ip )

def multiControllerNet():
net = Mininet( controller=RemoteController, switch=Switch)

print "*** Creating controllers"
c1 = net.addController( 'c1', port=6633 )
c2 = net.addController( 'c2', port=6634 )

# network 1
print "*** Creating switches for zone 1"
s1 = net.addSwitch( 's1' , "00:00:00:00:00:01", "10.0.0.1")
s2 = net.addSwitch( 's2', "00:00:00:00:00:02", "10.0.0.2" )
```

```
s3 = net.addSwitch( 's3' , "00:00:00:00:00:03", "10.0.0.3")
s4 = net.addSwitch( 's4', "00:00:00:00:00:04", "10.0.0.4" )
s5 = net.addSwitch( 's5', "00:00:00:00:00:05", "10.0.0.5")
s6 = net.addSwitch( 's6', "00:00:00:00:00:06", "10.0.0.6" )
s7 = net.addSwitch( 's7' , "00:00:00:00:00:07", "10.0.0.7")
s8 = net.addSwitch( 's8', "00:00:00:00:00:08", "10.0.0.8" )
s9 = net.addSwitch( 's9' , "00:00:00:00:00:09", "10.0.0.9")
s10 = net.addSwitch( 's10', "00:00:00:00:00:0a", "10.0.0.10" )

print "*** Creating hosts for zone 1"
senders1 = [ addHost( net, n ) for n in 11, 12 ]
alt_senders1 = [ addHost( net, n ) for n in 13, 14 ]  # senders
   after migration
receivers1 = [ addHost( net, n ) for n in 15, 16 ]
IDS = addHost( net, 21)
AppFW = addHost( net, 22)
DPI = addHost( net, 23)
IDS2 = addHost( net, 25)
AppFW2 = addHost( net, 24)

# network 2
print "*** Creating switches for zone 2"
s31 = net.addSwitch( 's31' , "00:00:00:00:00:1f", "10.0.0.31")
s32 = net.addSwitch( 's32', "00:00:00:00:00:20", "10.0.0.32" )
s33 = net.addSwitch( 's33' , "00:00:00:00:00:21", "10.0.0.33")
s34 = net.addSwitch( 's34', "00:00:00:00:00:22", "10.0.0.34" )
s35 = net.addSwitch( 's35', "00:00:00:00:00:23", "10.0.0.35")
s36 = net.addSwitch( 's36', "00:00:00:00:00:24", "10.0.0.36" )
s37 = net.addSwitch( 's37' , "00:00:00:00:00:25", "10.0.0.37")
s38 = net.addSwitch( 's38', "00:00:00:00:00:26", "10.0.0.38" )
s39 = net.addSwitch( 's39' , "00:00:00:00:00:27", "10.0.0.39")
s40 = net.addSwitch( 's40', "00:00:00:00:00:28", "10.0.0.40" )

print "*** Creating hosts for zone 2"
receiver2 = addHost( net, 45 )
IDS3 = addHost( net, 51)
```

```
AppFW3 = addHost( net, 52)
DPI3 = addHost( net, 53)

# linkage zone 1
print "*** Creating links in zone 1"
[ s1.linkTo( h ) for h in senders1 ]

s1.linkTo( s2 )
s2.linkTo( s3 )
s3.linkTo( s4 )
s4.linkTo( s5 )
s5.linkTo( s6 )

[ s6.linkTo( h ) for h in receivers1 ]

s4.linkTo( s7 )
s7.linkTo( s8 )
s8.linkTo( s9 )
s9.linkTo( s10 )

[ s9.linkTo( h ) for h in alt_senders1 ]

s2.linkTo( IDS )
s3.linkTo( AppFW )
s5.linkTo( DPI )
s7.linkTo( AppFW2 )
s8.linkTo( IDS2 )

# linkage inter-zone
print "*** link zone 1 zone 2"
s10.linkTo( s31 )

# linkage zone 2
print "*** Creating links in zone 1"
s31.linkTo( s32 )
s32.linkTo( s33 )
```

```
s33.linkTo( s34 )
s34.linkTo( s35 )
s35.linkTo( s36 )
s36.linkTo( s37 )
s35.linkTo( s38 )
s38.linkTo( s39 )
s39.linkTo( s40 )


s32.linkTo( IDS3 )
s33.linkTo( AppFW3 )
s34.linkTo( DPI3 )

s36.linkTo( receiver2 )

# run
print "*** Starting network"
net.build()
[ controller.start() for controller in c1, c2 ]
[ sw.start( [c1] ) for sw in s1, s2, s3 ,s4 ,s5, s6, s7, s8, s9,
   s10]
[ sw.start( [c2] ) for sw in s31, s32, s33 ,s34 ,s35, s36, s37,
   s38, s39, s40]

print "*** Running CLI"
CLI( net )

print "*** Stopping network"
net.stop()

if __name__ == '__main__':
setLogLevel( 'info' )  # for CLI output
multiControllerNet()
```

APPENDICES C

send packets.py

```
#!/usr/bin/env python
"""
This script sends a crafted packet marked with source 10.0.0.11
   and dst 10.0.0.15 tagged with AppID 2
by RaaverooK
"""
import array
import struct
import socket
import fcntl

from struct import *

SIOCGIFCONF = 0x8912  # define SIOCGIFCONF
BYTES = 4096          # define the byte size

# get_iface_list function definition
# this function will return array of all 'up' interfaces
def get_iface_list():
# create the socket object to get the interface list
sck = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# prepare the struct variable
names = array.array('B', '\0' * BYTES)
# the trick is to get the list from ioctl
bytelen = struct.unpack('iL', fcntl.ioctl(sck.fileno(),
   SIOCGIFCONF, struct.pack('iL', BYTES, names.buffer_info()[0])))
   [0]
# convert it to string
namestr = names.tostring()
# return the interfaces as array
return [namestr[i:i+32].split('\0', 1)[0] for i in range(0,
```

```
    bytelen, 32)]

# now, use the function to get the 'up' interfaces array
ifaces = get_iface_list()

# create the sending socket, bind on the first interface, usually
    in mininet for host h11 : h11-eth0
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
s.bind((ifaces[0], 50007))

# We're putting together an ethernet frame
# MAC and source address
src_addr = "\x00\x00\x00\x00\x00\x0b"
dst_addr = "\x00\x00\x00\x00\x00\x0f"
# the data
payload = ("["*30)+"PAYLOAD"+("]"*30)
# ethertype IP
ethertype = "\x08\x00"

# version, header length, total length, flag, ttl, proto (udp)
ip1 = "\x46\x00\x00\x63\x00\x8a\x40\x00\x40\x11"
ip_checksum = "\x22\xe3"
# source and dst ip
ip2 = "\x0a\x00\x00\x0b\x0a\x00\x00\x0f"
# app_id
ip_opt = "\x02\x02\x00\x02"
ip = ip1 + ip_checksum + ip2 + ip_opt

# port 50007 to 50007
UDP = "\xc3\x57\xc3\x57\x00\x40\x00\x00"
# send the packet through the socket
s.send(dst_addr+src_addr+ethertype+ip+UDP+payload)
```

APPENDICES D

receive packets.py

```
#!/usr/bin/python
"""
This script listens to incoming ip packets and display the src and
    dst when a packet is received
by RaaverooK
"""

import socket
from struct import *

#Convert a string of 6 characters of ethernet address into a dash
   separated hex string
def eth_addr (a) :
b = "%.2x-%.2x-%.2x-%.2x-%.2x-%.2x" % (ord(a[0]) , ord(a[1]) , ord
   (a[2]), ord(a[3]), ord(a[4]) , ord(a[5]))
return b

#create an PACKET , RAW SOCKET
#listens to all incoming ethernet packets
s = socket.socket( socket.AF_PACKET , socket.SOCK_RAW , socket.
   ntohs(0x0003))

# receive a packet
while True:
packet = s.recvfrom(65565)
#packet string from tuple
packet = packet[0]
#parse ethernet header
eth_length = 14
eth_header = packet[:eth_length]
eth = unpack('!6s6sH' , eth_header)
```

```
eth_protocol = socket.ntohs(eth[2])
#Parse IP packets (no ARP)
if eth_protocol == 8 :
#Parse IP header
#take first 24 characters for the ip header (classic header +
    options)
ip_header = packet[eth_length:24+eth_length]
#now unpack them
iph = unpack('!BBHHHBBH4s4sBBBB', ip_header)

protocol = iph[6]

# avoirding ICMP
if protocol != 1 :
s_addr = socket.inet_ntoa(iph[8]);
d_addr = socket.inet_ntoa(iph[9]);
#print "received packet"
print "received : ", s_addr, "  to : ", d_addr
```

APPENDICES E

middlebox.py

```
#!/usr/bin/python
"""
Script which reproduce the middlebox behavior :
- sniff packets
- resend them
by RaaverooK
"""
import array
import struct
import socket
import fcntl

from struct import *

SIOCGIFCONF = 0x8912  # define SIOCGIFCONF
BYTES = 4096          # Simply define the byte size

# get_iface_list function definition
# this function will return array of all 'up' interfaces
def get_iface_list():
# create the socket object to get the interface list
sck = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# prepare the struct variable
names = array.array('B', '\0' * BYTES)
# the trick is to get the list from ioctl
bytelen = struct.unpack('iL', fcntl.ioctl(sck.fileno(),
    SIOCGIFCONF, struct.pack('iL', BYTES, names.buffer_info()[0])))
    [0]
# convert it to string
namestr = names.tostring()
# return the interfaces as array
```

```
return [namestr[i:i+32].split('\0', 1)[0] for i in range(0,
    bytelen, 32)]

# now, use the function to get the 'up' interfaces array
ifaces = get_iface_list()

# create a socket to listen to arriving ethernet packets
s = socket.socket( socket.AF_PACKET , socket.SOCK_RAW , socket.
    ntohs(0x0800))

# create the sending socket which will be used to re-send the
    sniffed packets, mimicking a middlebox behavior
sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
# we bind it to the first interface up, usually in mininet for
    host h11 : h11-eth0
sock.bind((ifaces[0], 50007))

# receive a packet
while True:
packet = s.recvfrom(65565)
#packet string from tuple
packet = packet[0]
#parse ethernet header
eth_length = 14
eth_header = packet[:eth_length]
#unpack the info
eth = unpack('!6s6sH' , eth_header)
eth_protocol = socket.ntohs(eth[2])

#Parse IP packets (no ARP)
if eth_protocol == 8 :
#take first 24 characters for the ip header (regular header +
    options)
ip_header = packet[eth_length:24+eth_length]
#now unpack them
iph = unpack('!BBHHHBBH4s4sBBBB' , ip_header)
```

```
protocol = iph[6]
# avoiding ICMP
if protocol != 1 :
s_addr = socket.inet_ntoa(iph[8]);
d_addr = socket.inet_ntoa(iph[9]);
# print out the information about the packet
print "re-sent: ", s_addr, " to : ", d_addr
# resend it
sock.send(packet)
#print "packet re-sent"
```

APPENDICES F

Tests - Flow entries inside the OFSs

flow-entries in s1

Rule 1
- Match : from .11 to .15
- Action : put tag IDS1, route to s2

Rule 2
- Match : from .12 to .16
- Action : put tag IDS1, route to s2

Rule 3
- Match : from .12 to .45
- Action : put tag IDS1, route to s2

flow-entries in s2

Rule 1
- Match : eel tag IDS1
- Action : pop eel tag, route to port IDS1

Rule 2
- Match : inport IDS1, from .11 to .15
- Action : put tag AppFW1, route to s3

Rule 3
- Match : inport IDS1, from .12 to .16
- Action : put tag AppFW1, route to s3

Rule 4
- Match : inport IDS1, from .12 to .45
- Action : put tag AppFW1, route to s3

flow-entries in s3

Rule 1
- Match : eel tag AppFW1
- Action : pop eel tag, route to port AppFW1

Rule 2
- Match : inport AppFW1, from .11 to .15
- Action : route to s4

Rule 3
- Match : inport AppFW1, from .12 to .16
- Action : put tag DPI1, route to s4

Rule 4
- Match : inport AppFW1, from .12 to .45
- Action : put gtag “DPI”, route to s4

flow-entries in s4

Rule 1
- Match : eel tag DPI1
- Action : route to port s5

Rule 2
- Match : from .11 to .15
- Action : route to s5

Rule 3
- Match : gtag “DPI”, from .12 to .45
- Action : route to s5

flow-entries in s5

Rule 1
- Match : eel tag DPI1
- Action : pop eel tag, route to port DPI1

Rule 2

- Match : from .11 to .15
- Action : route to s6

Rule 3

- Match : inport DPI1, from .12 to .16
- Action : route to s6

flow-entries in s6

Rule 1

- Match : from .11 to .15
- Action : route to h15

Rule 2

- Match : from .12 to .16
- Action : route to h16

flow-entries in s7

Rule 1

- Match : eel tag AppFW2
- Action : pop eel tag, route to port AppFW2

Rule 2

- Match : inport AppFW2, from .11 to .15
- Action : route to s4

Rule 3

- Match : inport AppFW2, from .12 to .16
- Action : put tag DPI1, route to s4

Rule 4

- Match : gtag “DPI”, from .12 to .45
- Action : route to s8

flow-entries in s8

Rule 1

- Match : eel tag IDS2
- Action : pop eel tag, route to port IDS2

Rule 2

- Match : inport IDS2, from .11 to .15
- Action : put tag AppFW2, route to s7

Rule 3

- Match : inport IDS2, from .12 to .16
- Action : put tag AppFW2, route to s7

Rule 4

- Match : gtag "DPI", from .12 to .45
- Action : route to s9

flow-entries in s9

Rule 1

- Match : from .11 to .15
- Action : put tag IDS2, route to s8

Rule 2

- Match : from .12 to .16
- Action : put tag IDS2, route to s8

Rule 3

- Match : gtag "DPI", from .12 to .45
- Action : route to s10

Rule 4

- Match : from .12 to .45
- Action : put gtag "IDS", route to s10

flow-entries in s10

Rule 1

- Match : gtag “DPI”, from .12 to .45
- Action : route to s31

Rule 2

- Match : gtag “IDS”, from .12 to .45
- Action : route to s31

flow-entries in s31

Rule 1

- Match : gtag “IDS”, from .12 to .45
- Action : pop eel tag, put eel tag IDS3, route to s32

Rule 2

- Match : gtag “DPI”, from .12 to .45
- Action : pop eel tag, put eel tag DPI3, route to s32

flow-entries in s32

Rule 1

- Match : eel tag IDS3
- Action : pop eel tag, route to IDS3

Rule 2

- Match : inport IDS3, from .12 to .45
- Action : put eel tag AppFW3, route to s33

Rule 3

- Match : eel tag DPI3,
- Action : route to s33

flow-entries in s33

Rule 1

- Match : eel tag AppFW3
- Action : pop eel tag, route to AppFW3

Rule 2

- Match : inport AppFW3, from .12 to .45
- Action : put eel tag DPI3, route to s34

Rule 3

- Match : eel tag DPI3,
- Action : route to s34

flow-entries in s34

Rule 1

- Match : eel tag DPI3
- Action : pop eel tag, route to DPI3

Rule 2

- Match : inport DPI3, from .12 to .45
- Action : put eel tag DPI3, route to s35

flow-entries in s35

Rule 1

- Match : from .12 to .45
- Action : route to s36

flow-entries in s36

Rule 1

- Match : from .12 to .45
- Action : route to s45